C++ Annotations Version 8.1.0~pre2

Frank B. Brokken
Center of Information Technology,
University of Groningen
Nettelbosje 1,
P.O. Box 11044,
9700 CA Groningen
The Netherlands
Published at the University of Groningen
ISBN 90 367 0470 7

Abstract

This document is intended for knowledgeable users of **C** (or any other language using a **C**-like grammar, like **Perl** or **Java**) who would like to know more about, or make the transition to, **C++**. This document is the main textbook for Frank's **C++** programming courses, which are yearly organized at the University of Groningen. The **C++** Annotations do not cover all aspects of **C++**, though. In particular, **C++**'s basic grammar is not covered when equal to **C**'s grammar. Any basic book on **C** may be consulted to refresh that part of **C++**'s grammar.

If you want a **hard-copy version of the C++ Annotations**: printable versions are available in postscript, pdf and other formats in

`http://sourceforge.net/projects/cppannotations/`,

in files having names starting with `cplusplus` (A4 paper size). Files having names starting with 'cplusplusus' are intended for the US *legal* paper size.

The latest version of the **C++** Annotations in html-format can be browsed at:

`http://cppannotations.sourceforge.net/`
and/or at
`http://www.icce.rug.nl/documents/`

Contents

Chapter 1

Overview Of The Chapters

The chapters of the **C++** Annotations cover the following topics:

- Chapter 1: This overview of the chapters.
- Chapter 2: A general introduction to **C++**.
- Chapter 3: A first impression: differences between **C** and **C++**.
- Chapter 4: Name Spaces: how to avoid name collisions.
- Chapter 5: The 'string' data type.
- Chapter 6: The **C++** I/O library.
- Chapter 7: The 'class' concept: structs having functions. The 'object' concept: variables of a `class`.
- Chapter 8: Allocation and returning unused memory: `new`, `delete`, and the function `set_new_handler()`.
- Chapter 9: Exceptions: handle errors where appropriate, rather than where they occur.
- Chapter 10: Give your own meaning to operators.
- Chapter 11: Static data and functions: members of a class not bound to objects.
- Chapter 12: Abstract Containers to put stuff into.
- Chapter 13: Building classes upon classes: setting up class hierarcies.
- Chapter 14: Changing the behavior of member functions accessed through base class pointers.
- Chapter 15: Gaining access to private parts: friend functions and classes.
- Chapter 16: Classes having pointers to members: pointing to locations inside objects.
- Chapter 17: Constructing classes and enums within classes.
- Chapter 18: The Standard Template Library.
- Chapter 19: The STL generic algorithms.
- Chapter 20: Function templates: using *molds* for type independent functions.
- Chapter 21: Class templates: using *molds* for type independent classes.
- Chapter 22: Advanced Template Use: programming the compiler.
- Chapter 23: Several examples of programs written in **C++**.

Chapter 2

Introduction

This document offers an introduction to the **C++** programming language. It is a guide for **C/C++** programming courses, yearly presented by Frank at the University of Groningen. This document is not a complete **C/C++** handbook, as much of the **C**-background of **C++** is not covered. Other sources should be referred to for that (e.g., the Dutch book *De programmeertaal C*, Brokken and Kubat, University of Groningen, 1996) or the `on-line book`[1] suggested to me by George Danchev (danchev at spnet dot net).

The reader should be forwarned that extensive knowledge of the **C** programming language is actually assumed. The **C++** Annotations continue where topics of the **C** programming language end, such as pointers, basic flow control and the construction of functions.

The version number of the **C++** Annotations (currently 8.1.0~pre2) is updated when the contents of the document change. The first number is the major number, and will probably not be changed for some time: it indicates a major rewriting. The middle number is increased when new information is added to the document. The last number only indicates small changes; it is increased when, e.g., series of typos are corrected.

The **C++ Annotations** were typeset using the `yodl`[3] formatting system.

All correspondence concerning suggestions, additions, improvements or changes to this document should be directed to the author:

Frank B. Brokken
Center of Information Technology,
University of Groningen
Nettelbosje 1,
P.O. Box 11044,
9700 CA Groningen
The Netherlands
(email: f.b.brokken@rug.nl)

In this chapter an overview of **C++**'s defining features is presented. A few extensions to **C** are reviewed and the concepts of object based and object oriented programming (OOP) are briefly introduced.

[1] http://publications.gbdirect.co.uk/c_book/
[2] http://www.gnu.org/licenses/
[3] http://yodl.sourceforge.net

2.1 What's new in the C++ Annotations

This section is modified when the first or second part of the version number changes (and sometimes for the third part as well).

- Version 8.1.0 provides an update of many of the sections labeled **C++0x**. Sections showing `C++-0x` may also mention the `g++` version in which the new feature will be made available, using `?` if this is as yet unknown. Many inconsistencies that have crept into the text and examples have been removed.

- Version 8.0.0 was released as a result of the upcoming new **C++** standard[4] becoming (partially) available in the Gnu `g++` compiler[5]. Not all new elements of the new standard (informally called the C++0x standard) are available right now, and new subreleases of the **C++** Annotations will appear once more elements become implemented in the `g++` compiler. In section 2.2.3 the way to activate the new standard is shown, and new sections covering elements of the new standard show `C++0x` in their section-titles.
  Furthermore, two new chapters were added: the STL chapter is now split in two. The STL chapter now covers the STL except for the *Generic Algorithms* which are now discussed in a separate chapter. Name spaces, originally covered by the introductory chapter are now also covered in a separate chapter.

- Version 7.3.0 adds a section about overloading operators outside of the common context of classes (section 10.11).

- Version 7.2.0 describes the implementation of polymorphism for classes inheriting from multiple base classes defining virtual member functions (section 14.9) and adds two new sections in the concrete examples chapter: Section 23.1 discusses the problem how to distinguish *lvalues* from *rvalues* with `operator[]()`, section 23.9.3 discusses in the context of the Bisonc++ parser generator how to use polymorphism instead of a union to define different types of semantic values. As usual, several typos were repaired and various other improvements were made.

- Version 7.1.0 adds a description of the `type_info::before()` member (cf. section 14.5.2). Furthermore, several typographical corrections were made.

- Version 7.0.1. was released shortly after releasing version 7.0.0, as the result of very extensive feedback received by Eric S. Raymond (esr at thyrsus dot com) and Edward Welbourne (eddy at chaos dot org dot uk). Considering the extent of the received feedback, it's appropriate to mention explicitly the sub-sub-release here. Many textual changes were made and section 5.2.4 was completely reorganized.

- Version 7.0.0 comes with a new chapter discussing advanced template applications. Moreover, the general terminology used with templates has evolved. 'Templates' are now considered a core concept, which is reflected by the use of 'templates' as a noun, rather than an adjective. So, from now on it is 'class template' rather than 'template class'. The addition of another chapter, together with the addition of several new sections to existing chapters as well as various rewrites of existing sections made it appropriate to upgrade to the next major release. The newly added chapter does not aim at concrete examples of templates. Instead it discusses possibilities of templates beyond the basic function and class templates. In addition to this new chapter, several new sections were added: section 7.7 introduces *local classes*; section 8.1.4 discusses the *placement new operator*; section 13.6.1 discusses how to make available some members of privately inherited classes and section 13.8 discusses how objects created by `new[]` can be initialized by non-default constructors. In addition to all this, Elwin Dijck (e dot dijck at gmail dot com), one of the students of the 2006-2007 edition of the **C++** course, did a *magnificent* job by converting *all* images to vector graphics (in the process prompting me to start using vector graphics as well :-). Thanks, Elwin for a job well done!

- Version 6.5.0 changed `unsigned` into `size_t` where appropriate, and explicitly mentioned `int`-derived types like `int16_t`. In-class member function definitions were moved out of (below) their

[4] `http://en.wikipedia.org/wiki/C++0x`
[5] http://gcc.gnu.org/projects/cxx0x.html

class definitions as `inline` defined members. A paragraphs about implementing pure virtual member functions was added. Various bugs and compilation errors were fixed.

- Version 6.4.0 added a new section (22.1.2) further discussing the use of the `template` keyword to distinguish types nested under class templates from template members. Furthermore, *Sergio Bacchi* `s dot bacchi at gmail dot com` did an impressive job when translating the Annotations into Portuguese. His translation (which may lag a distribution or two behind the latest verstion of the Annotations) may also be retrieved from the `contributions/` subdirectory in the `c++-annotations_X.Y.Z.tar.gz` archive at `http://sourceforge.net/projects/cppannotations/`

- Version 6.3.0 added new sections about anonymous objects (section 7.5.1) and type resolution with class templates (section 22.1.3). Also the description of the template parameter deduction algorithm was rewritten (cf. section 20.3.4) and numerous modifications required because of the compiler's closer adherence to the **C++** standard were realized, among which exception rethrowing from constructor and destructor function try blocks. Also, all textual corrections received from readers since version 6.2.4 were processed.

- In version 6.2.4 many textual improvements were realized. I received extensive lists of typos and suggestions for clarifications of the text, in particular from Nathan Johnson and from Jakob van Bethlehem. Equally valuable were suggestions I received from various other readers of the **C++** annotations: all were processed in this release. The **C++** content matter of this release was not substantially modified, compared to version 6.2.2.

- Version 6.2.2 offers improved implementations of the configurable class templates (sections 23.8.3 and 23.8.4).

- Version 6.2.0 was released as an Annual Update, by the end of May, 2005. Apart from the usual typo corrections several new sections were added and some were removed: in the Exception chapter (9) a section was added covering the standard exceptions and their meanings; in the chapter covering static members (11) a section was added discussing `static const` data members; and the final chapter (23) covers configurable class templates using *local context structs* (replacing the previous `ForEach`, `UnaryPredicate` and `BinaryPredicate` classes). Furthermore, the final section (covering a **C++** parser generator) now uses **bisonc++**, rather than the old (and somewhat outdated) **bison++** program.

- Version 6.1.0 was released shortly after releasing 6.0.0. Following suggestions received from Leo Razoumov`<LEOR@winmain.rutgers.edu>` and Paulo Tribolet, and after receiving many, many useful suggestions and extensive help from Leo, navigatable `.pdf` files are from now on distributed with the **C++** Annotations. Also, some sections were slightly adapted.

- Version 6.0.0 was released after a full update of the text, removing many inconsistencies and typos. Since the update effected the Annotation's full text an upgrade to a new major version seemed appropriate. Several new sections were added: overloading binary operators (section 10.6); throwing exceptions in constructors and destructors (section 9.11); function `try`-blocks (section 9.10); calling conventions of static and global functions (section 11.2.1) and virtual constructors (section 14.11). The chapter on templates was completely rewritten and split into two separate chapters: chapter 20 discusses the syntax and use of template *functions*; chapter 21 discusses template *classes*. Various concrete examples were modified; new examples were included as well (chapter 23).

- In version 5.2.4 the description of the *random_shuffle* generic algorithm (section 19.1.39) was modified.

- In version 5.2.3 section 2.5.11 on local variables was extended and section 2.5.4 on function overloading was modified by explicitly discussing the effects of the **const** modifier with overloaded functions. Also, the description of the `compare()` function in chapter 5 contained an error, which was repaired.

- In version 5.2.2 a leftover in section 10.4 from a former version was removed and the corresponding text was updated. Also, some minor typos were corrected.

- In version 5.2.1 various typos were repaired, and some paragraphs were further clarified. Furthermore, a section was added to the *template* chapter (chapter 20), about creating several `iterator` types. This topic was further elaborated in chapter 23, where the section about the construction of a reverse iterator (section 23.6) was completely rewritten. In the same chapter, a *universal text to anything convertor* is discussed (section 23.7). Also, `LaTeX`, `PostScript` and `PDF` versions fitting the *US-letter* paper size are now available as `cplusplus`**us** versions: `cplusplusus.latex`, `cplusplusus.ps` and `cplusplus.pdf`. The *A4-paper* size is of course kept, and remains to be available in the `cplusplus.latex`, `cplusplus.ps` and `cpluspl.pdf` files.

- Version 5.2.0 was released after adding a section about the `mutable` keyword (section 7.8), and after thoroughly changing the discussion of the `Fork()` abstract base class (section 23.4). All examples should now be up-to-date with respect to the use of the `std` namespace.

- However, in the meantime the Gnu `g++` compiler version 3.2 was released[6]. In this version extensions to the abstract containers (see chapter 12) like the hash_map were placed in a separate namespace, `__gnu_cxx`. This namespace should be used when using these containers. However, this may break compilations of sources with `g++`, version 3.0. In that case, a compilation can be performed conditionally to the 3.2 and the 3.0 compiler version, defining `__gnu_cxx` for the 3.2 version. Alternatively, the *dirty trick*

  ```
  #define __gnu_cxx std
  ```

  can be placed just before header files in which the `__gnu_cxx` namespace is used. This might eventually result in name-collisions, and it's a dirty trick by any standards, so please don't tell anybody I wrote this down.

- Version 5.1.1 was released after modifying the sections related to the `fork()` system call in chapter 23. Under the ANSI/ISO standard many of the previously available extensions (like `procbuf`, and `vform()`) applied to streams were discontinued. Starting with version 5.1.1. ways of constructing these facilities under the ANSI/ISO standard are discussed in the **C++** Annotations. I consider the involved subject sufficiently complex to warrant the upgrade to a new subversion.

- With the advent of the Gnu `g++` compiler version 3.00, a more strict implementation of the ANSI/ISO **C++** standard became available. This resulted in version 5.1.0 of the Annotations, appearing shortly after version 5.0.0. In version 5.1.0 chapter 6 was modified and several cosmetic changes took place (e.g., removing `class` from template type parameter lists, see chapter 20). Intermediate versions (like 5.0.0a, 5.0.0b) were not further documented, but were mere intermediate releases while approaching version 5.1.0. Code examples will gradually be adapted to the new release of the compiler.

  In the meantime the reader should be prepared to insert

  ```
  using namespace std;
  ```

  in many code examples, just beyond the `#include` preprocessor directives as a temporary measure to make the example accepted by the compiler.

- New insights develop all the time, resulting in version 5.0.0 of the Annotations. In this version a lot of old code was cleaned up and typos were repaired. According to current standard, *namespaces* are required in **C++** programs, so they are introduced now very early (in section 3.1.1) in the Annotations. A new section about using external programs was added to the Annotations (and removed again in version 5.1.0), and the new `stringstream` class, replacing the `strstream` class is now covered too (sections 6.4.3 and 6.5.3). Actually, the chapter on input and output was completely rewritten. Furthermore, the operators `new` and `delete` are now discussed in chapter 8, where they fit better than in a chapter on classes, where they previously were discussed. Chapters were moved, split and reordered, so that subjects could generally be introduced without forward references. Finally, the `html`, PostScript and pdf versions of the **C++** Annotations now contain an index (sigh of relief ?) All in, considering the volume and nature of the modifications, it seemed right to upgrade to a full major version. So here it is.

  Considering the volume of the Annotations, I'm sure there will be typos found every now and then. Please do not hesitate to send me mail containing any mistakes you find or corrections you would like to suggest.

[6] `http://www.gnu.org`

- In release 4.4.1b the pagesize in the LaTeX file was defined to be din A4. In countries where other pagesizes are standard the default pagesize might be a better choice. In that case, remove the a4paper,twoside option from cplusplus.tex (or cplusplus.yo if you have yodl installed), and reconstruct the Annotations from the *TeX*-file or Yodl-files.

  The Annotations mailing lists was stopped at release 4.4.1d. From this point on only minor modifications were expected, which are not anymore generally announced.

  At some point, I considered version 4.4.1 to be the final version of the **C++** Annotations. However, a section on special I/O functions was added to cover unformatted I/O, and the section about the string datatype had its layout improved and was, due to its volume, given a chapter of its own (chapter 5). All this eventually resulted in version 4.4.2.

  Version 4.4.1 again contains new material, and reflects the ANSI/ISO[7] standard (well, I try to have it reflect the ANSI/ISO standard). In version 4.4.1. several new sections and chapters were added, among which a chapter about the *Standard Template Library* (STL) and *generic algorithms*.

  Version 4.4.0 (and subletters) was a mere construction version and was never made available.

  The version 4.3.1a is a precursor of 4.3.2. In 4.3.1a most of the typos I've received since the last update have been processed. In version 4.3.2 extra attention was paid to the syntax for function addresses and pointers to member functions.

  The decision to upgrade from version 4.2.* to 4.3.* was made after realizing that the lexical scanner function yylex() can be defined in the scanner class that is derived from yyFlexLexer. Under this approach the yylex() function can access the members of the class derived from yyFlexLexer as well as the public and protected members of yyFlexLexer. The result of all this is a clean implementation of the rules defined in the flex++ specification file.

  The upgrade from version 4.1.* to 4.2.* was the result of the inclusion of section 3.4.1 about the **bool** data type in chapter 3. The distinction between differences between **C** and **C++** and extensions of the **C** programming languages is (albeit a bit fuzzy) reflected in the introduction chapter and the chapter on first impressions of **C++**: The introduction chapter covers some differences between **C** and **C++**, whereas the chapter about first impressions of **C++** covers some extensions of the **C** programming language as found in **C++**.

  Major version 4 is a major rewrite of the previous version 3.4.14. The document was rewritten from SGML to Yodl and many new sections were added. All sections got a tune-up. The distribution basis, however, hasn't changed: see the introduction.

  Modifications in versions 1.*.*, 2.*.*, and 3.*.* (replace the stars by any applicable number) were not logged.

  Subreleases like 4.4.2a etc. contain bugfixes and typographical corrections.

2.2 C++'s history

The first implementation of **C++** was developed in the 1980s at the AT&T Bell Labs, where the Unix operating system was created.

C++ was originally a 'pre-compiler', similar to the preprocessor of **C**, converting special constructions in its source code to plain **C**. Back then this code was compiled by a standard **C** compiler. The 'pre-code', which was read by the **C++** pre-compiler, was usually located in a file with the extension .cc, .C or .cpp. This file would then be converted to a **C** source file with the extension .c, which was thereupon compiled and linked.

The nomenclature of **C++** source files remains: the extensions .cc and .cpp are still used. However, the preliminary work of a **C++** pre-compiler is nowadays usually performed during the actual compilation process. Often compilers determine the language used in a source file from its extension. This holds true for Borland's and Microsoft's **C++** compilers, which assume a **C++** source for an extension .cpp. The Gnu compiler g++, which is available on many Unix platforms, assumes for **C++** the extension .cc.

[7] ftp://research.att.com/dist/c++std/WP/

The fact that **C++** used to be compiled into **C** code is also visible from the fact that **C++** is a superset of **C**: **C++** offers the full **C** grammar and supports all **C**-library functions, and adds to this features of its own. This makes the transition from **C** to **C++** quite easy. Programmers familiar with **C** may start 'programming in **C++**' by using source files having extensions `.cc` or `.cpp` instead of `.c`, and may then comfortably slip into all the possibilities offered by **C++**. No abrupt change of habits is required.

2.2.1 History of the C++ Annotations

The original version of the **C++** Annotations was written by Frank Brokken and Karel Kubat in Dutch using `LaTeX`. After some time, Karel rewrote the text and converted the guide to a more suitable format and (of course) to English in september 1994.

The first version of the guide appeared on the net in october 1994. By then it was converted to `SGML`.

Gradually new chapters were added, and the contents were modified and further improved (thanks to countless readers who sent us their comment).

In major version four Frank added new chapters and converted the document from SGML to `yodl`[8].

> The **C++** Annotations are freely distributable. Be sure to read the `legal notes`[9].
>
> **Reading the annotations beyond this point implies that you are aware of these notes and that you agree with them.**

If you like this document, tell your friends about it. Even better, let us know by sending email to `Frank`[10].

In the Internet, many useful hyperlinks exist to **C++**. Without even suggesting completeness (and without being checked regularly for existence: they might have died by the time you read this), the following might be worthwhile visiting:

- `http://www.cplusplus.com/ref/`: a reference site for **C++**.
- `http://www.csci.csusb.edu/dick/c++std/cd2/index.html`: offers a version of the 1996 working paper of the **C++** ANSI/ISO standard.

2.2.2 Compiling a C program using a C++ compiler

Prospective **C++** programmers should realize that **C++** is not a perfect superset of **C**. There are some differences you might encounter when you simply rename a file to a file having the extension `.cc` and run it through a **C++** compiler:

- In **C**, `sizeof('c')` equals `sizeof(int)`, `'c'` being any ASCII character. The underlying philosophy is probably that `chars`, when passed as arguments to functions, are passed as integers anyway. Furthermore, the **C** compiler handles a character constant like `'c'` as an integer constant. Hence, in **C**, the function calls

  ```
  putchar(10);
  ```

  and

  ```
  putchar('\n');
  ```

[8] http://yodl.sourceforge.net
[9] legal.shtml
[10] mailto:f.b.brokken@rug.nl

are synonymous.

By contrast, in **C++**, `sizeof('c')` is always 1 (but see also section 3.4.2). An `int` is still an `int`, though. As we shall see later (section 2.5.4), the two function calls

```
somefunc(10);
```

and

```
somefunc('\n');
```

may be handled by different functions: **C++** distinguishes functions not only by their names, but also by their argument types, which are different in these two calls. The former using an `int` argument, the latter a `char`.

- **C++** requires very strict prototyping of external functions. E.g., in **C** a prototype like

  ```
  void func();
  ```

  means that a function `func()` exists, returning no value. The declaration doesn't specify which arguments (if any) are accepted by the function.

  However, in **C++** the above declaration means that the function `func()` does *not* accept any arguments at all. Any arguments passed to it will result in a compile-time error.

  Note that the keyword `extern` is not required when declaring functions. A function definition becomes a function declaration simply by replacing a function's body by a semicolon. The keyword `extern` *is* required, though, when declaring variables.

2.2.3 Compiling a C++ program

To compile a **C++** program, a **C++** compiler is required. Considering the free nature of this document, it won't come as a surprise that a *free compiler* is suggested here. The Free Software Foundation (FSF) provides at `http://www.gnu.org` a free **C++** compiler which is, among other places, also part of the Debian (`http://www.debian.org`) distribution of Linux (`http://www.linux.org`).

The upcoming C++0x standard has not yet fully been implemented in the `g++` compiler. Unless indicated otherwsie, all features of the C++0x standard covered by the **C++** Annotations are available in `g++ 4.4`, unless indicated otherwise. To use these features the compiler flag `-std=c++0x` must currently be provided. It is assumed that this flag is used when compiling the examples given by the Annotations. The features of the C++0x standard may or may not be available in `g++` versions older than 4.4.

2.2.3.1 C++ under MS-Windows

For MS-Windows Cygnus (`http://sources.redhat.com/cygwin`) provides the foundation for installing the *Windows port* of the Gnu `g++` compiler.

When visiting the above URL to obtain a free `g++` compiler, click on `install now`. This will download the file `setup.exe`, which can be run to install `cygwin`. The software to be installed can be downloaded by `setup.exe` from the internet. There are alternatives (e.g., using a CD-ROM), which are described on the Cygwin page. Installation proceeds interactively. The offered defaults are sensible and should be accepted unless you have reasons to divert.

The most recent Gnu `g++` compiler can be obtained from `http://gcc.gnu.org`. If the compiler that is made available in the Cygnus distribution lags behind the latest version, the sources of the latest version can be downloaded after which the compiler can be built using an already available compiler. The compiler's webpage (mentioned above) contains detailed instructions on how to proceed. In our experience building a new compiler within the Cygnus environment works flawlessly.

2.2.3.2 Compiling a C++ source text

Generally the following command can be used to compile a **C++** source file 'source.cc':

```
g++ source.cc
```

This produces a binary program (a.out or a.exe). If the default name is inappropriate, the name of the executable can be specified using the -o flag (here producing the program source):

```
g++ -o source source.cc
```

If a mere compilation is required, the compiled module can be produced using the -c flag:

```
g++ -c source.cc
```

This generates the file source.o, which can later on be linked to other modules. As pointed out, provide the compiler option –std=c++0x (note: two dashes). to activate the features of the C++0x standard.

C++ programs quickly become too complex to maintain 'by hand'. With all serious programming projects program maintenance tools are used. Usually the standard make program is be used to maintain **C++** programs, but good alternatives exist, like the icmake[11] program maintenance utility, ccbuild[12] or lake[13]

It is strongly advised to start using maintenance utilities early in the study of **C++**.

2.3 C++: advantages and claims

Often it is said that programming in **C++** leads to 'better' programs. Some of the claimed advantages of **C++** are:

- New programs would be developed in less time because old code can be reused.
- Creating and using new data types would be easier than in **C**.
- The memory management under **C++** would be easier and more transparent.
- Programs would be less bug-prone, as **C++** uses a stricter syntax and type checking.
- 'Data hiding', the usage of data by one program part while other program parts cannot access the data, would be easier to implement with **C++**.

Which of these allegations are true? Originally, our impression was that the **C++** language was somewhat overrated; the same holding true for the entire object-oriented programming (OOP) approach. The enthusiasm for the **C++** language resembles the once uttered allegations about Artificial-Intelligence (AI) languages like Lisp and Prolog: these languages were supposed to solve the most difficult AI-problems 'almost without effort'. New languages are often oversold: in the end, each problem can be coded in any programming language (say BASIC or assembly language). The advantages and disadvantages of a given programming language aren't in 'what you can do with them', but rather in 'which tools the language offers to implement an efficient and understandable solution to a programming problem'. Often these tools take the form of syntactic *restrictions*, enforcing or promoting certain constructions or which simply suggest intentions by applying or 'embracing' such syntactic forms. Rather than a long list of plain assembly instructions we now use flow control statements, functions, objects or even

[11] http://icmake.sourceforge.net/
[12] http://ccbuild.sourceforge.net/
[13] http://nl.logilogi.org/MetaLogi/LaKe

(with **C++**) so-called *templates* to structure and organize code and to express oneself 'eloquently' in the language of one's choice.

Concerning the above allegations of **C++**, we support the following, however.

- The development of new programs while existing code is reused can also be implemented in **C** by, e.g., using function libraries. Functions can be collected in a library and need not be re-invented with each new program. **C++**, however, offers specific syntax possibilities for code reuse, apart from function libraries (see chapters 13 and 20).
- Creating and using new data types is certainly possible in **C**; e.g., by using `structs`, `typedefs` etc.. From these types other types can be derived, thus leading to `structs` containing `structs` and so on. In **C++** these facilities are augmented by defining data types which are completely 'self supporting', taking care of, e.g., their memory management automatically (without having to resort to an independently operating memory management system as used in, e.g., **Java**).
- In **C++** memory management can in principle be either as easy or as difficult as it is in C. Especially when dedicated **C** functions such as `xmalloc` and `xrealloc` are used (allocating the memory or aborting the program when the memory pool is exhausted). However, with functions like `malloc` it is easy to err. Frequently errors in **C** programs can be traced back to miscalculations when using `malloc`. Instead, **C++** offers facilities to allocate memory in a somewhat safer way, using its `operator new`.
- Concerning 'bug proneness' we can say that **C++** indeed uses stricter type checking than **C**. However, most modern **C** compilers implement 'warning levels'; it is then the programmer's choice to disregard or get rid of the warnings. In **C++** many of such warnings become fatal errors (the compilation stops).
- As far as 'data hiding' is concerned, **C** does offer some tools. E.g., where possible, local or `static` variables can be used and special data types such as `structs` can be manipulated by dedicated functions. Using such techniques, data hiding can be implemented even in **C**; though it must be admitted that **C++** offers special syntactic constructions, making it far easier to implement 'data hiding' (and more in general: 'encapsulation') in **C++** than in **C**.

C++ in particular (and OOP in general) is of course not *the* solution to all programming problems. However, the language *does* offer various new and elegant facilities which are worth investigating. At the downside, the level of grammatical complexity of **C++** has increased significantly as compared to **C**. This may be considered a serious drawback of the language. Although we got used to this increased level of complexity over time, the transition was neither fast nor painless.

With the **C++** Annotations we hope to help the reader when transiting from **C** to **C++** by focusing on the additions of **C++** as compared to **C** and by leaving out plain **C**. It is our hope that you like this document and may benefit from it.

Enjoy and good luck on your journey into **C++**!

2.4 What is Object-Oriented Programming?

Object-oriented (and object-based) programming propagates a slightly different approach to programming problems than the strategy usually used in **C** programs. In **C** programming problems are usually solved using a 'procedural approach': a problem is decomposed into subproblems and this process is repeated until the subtasks can be coded. Thus a conglomerate of functions is created, communicating through arguments and variables, global or local (or `static`).

In contrast (or maybe better: in addition) to this, an object-based approach identifies the **keywords** used in a problem statement. These keywords are then depicted in a diagram where arrows are drawn between those keywords to depict an internal hierarchy. The keywords become the objects in the implementation and the hierarchy defines the relationship between these objects. The term object is used here to describe a limited, well-defined structure, containing all information about an entity: data types

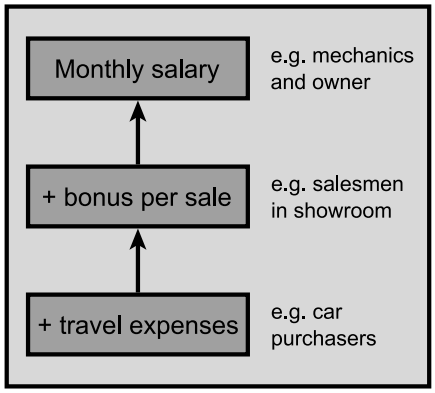

Figure 2.1: Hierarchy of objects in the salary administration.

and functions to manipulate the data. As an example of an object oriented approach, an illustration follows:

> The employees and owner of a car dealer and auto garage company are paid as follows. First, mechanics who work in the garage are paid a certain sum each month. Second, the owner of the company receives a fixed amount each month. Third, there are car salesmen who work in the showroom and receive their salary each month plus a bonus per sold car. Finally, the company employs second-hand car purchasers who travel around; these employees receive their monthly salary, a bonus per bought car, and a restitution of their travel expenses.

When representing the above salary administration, the keywords could be mechanics, owner, salesmen and purchasers. The properties of such units are: a monthly salary, sometimes a bonus per purchase or sale, and sometimes restitution of travel expenses. When analyzing the problem in this manner we arrive at the following representation:

- The owner and the mechanics can be represented by identical types, receiving a given salary per month. The relevant information for such a type would be the monthly amount. In addition this object could contain data as the name, address and social security number.
- Car salesmen who work in the showroom can be represented as the same type as above but with some *extra* functionality: the number of transactions (sales) and the bonus per transaction.

  In the hierarchy of objects we would define the dependency between the first two objects by letting the car salesmen be 'derived' from the owner and mechanics.
- Finally, there are the second-hand car purchasers. These share the functionality of the salesmen except for travel expenses. The additional functionality would therefore consist of the expenses made and this type would be derived from the salesmen.

The hierarchy of the identified objects are further illustrated in Figure 2.1.

The overall process in the definition of a hierarchy such as the above starts with the description of the most simple type. Traditionally (and still in vogue with some popular object oriented languages) more complex types are then derived from the basic set, with each derivation adding a little extra functionality. From these derived types, more complex types can be derived *ad infinitum*, until a representation of the entire problem can be made. Over the years, however, this approach has become less popular in **C++** as it typically results in overly tight *coupling*, which in turns *reduces* rather than enhances the understanding, maintainability and testability of complex programs. In **C++** object oriented program more and more favors small, easy to understand hierarchies, limited coupling and a developmental process where *design patterns* (cf. *Gamma et al.* (1995)) play a central role.

Nonetheless, in **C++** *classes* are frequently used to define the characteristics of *objects*. Classes contain the necessary functionality to do useful things. Classes generally do not offer all their functionality (and typically *none* of their data) to objects of other classes. As we will see, classes tend to *hide* their

properties in such a way that they are not directly modifiable by the outside world. Instead, dedicated functions are used to reach or modify the properties of objects. Thus class-type objects are able to uphold their own integrity. The core concept here is *encapsulation* of which *data hiding* is just an example. These concepts will be further explained in chapter 7.

2.5 Differences between C and C++

In this section some examples of **C++** code are shown. Some differences between **C** and **C++** are highlighted.

2.5.1 The function main()

In **C++** there are but two variants of the function `main()`: `int main()` and `int main(int argc, char **argv)`. Notes:

- The return type of `main` is `int`, and *not* `void`.
- It is *not* required to use an explicit `return` statement at the end of `main`. If omitted `main` returns 0.
- The 'third `char **envp` parameter' is not defined by the **C++** standard and should be avoided. Instead, the global variable `extern char **environ` should be declared providing access to the program's environment variables. Its final element has the value 0.

2.5.2 End-of-line comment

According to the ANSI definition, 'end of line comment' is implemented in the syntax of **C++**. This comment starts with `//` and ends at the end-of-line marker. The standard **C** comment, delimited by `/*` and `*/` can still be used in **C++**:

```
int main()
{
    // this is end-of-line comment
    // one comment per line

    /*
        this is standard-C comment, covering
        multiple lines
    */
}
```

Despite the example, it is advised *not* to use **C** type comment inside the body of **C++** functions. Sometimes existing code must temporarily be suppressed, e.g., for testing purposes. In those cases it's very practical to be able to use standard **C** comment. If such suppressed code itself contains such comment, it would result in nested comment-lines, resulting in compiler errors. Therefore, the rule of thumb is not to use **C** type comment inside the body of **C++** functions (alternatively, `#if 0` until `#endif` pair of preprocessor directives could of course also be used).

2.5.3 Strict type checking

C++ uses very strict type checking. A prototype must be known for each function before it is called, and the call must match the prototype. The program

```
int main()
{
    printf("Hello World\n");
}
```

often compiles under **C**, albeit with a warning that `printf()` is an unknown function. But **C++** compilers (should) fail to produce code in such cases. The error is of course caused by the missing `#include <stdio.h>` (which in **C++** is more commonly included as `#include <cstdio>` directive).

And while we're at it: as we've seen in **C++** `main` *always* uses the `int` return value. Although it is possible to define `int main()` without explicitly defining a return statement, within `main` it is not possible to use a `return` statement without an explicit `int`-expression. For example:

```
int main()
{
    return;     // won't compile: expects int expression, e.g.
                // return 1;
}
```

2.5.4 Function Overloading

In **C++** it is possible to define functions having identical names but performing different actions. The functions must differ in their parameter lists (and/or in their `const` attribute). An example is given below:

```
#include <stdio.h>

void show(int val)
{
    printf("Integer: %d\n", val);
}

void show(double val)
{
    printf("Double: %lf\n", val);
}

void show(char const *val)
{
    printf("String: %s\n", val);
}

int main()
{
    show(12);
    show(3.1415);
    show("Hello World\n!");
}
```

In the above program three functions `show` are defined, only differing in their parameter lists, expecting an `int`, `double` and `char *`, respectively. The functions have identical names. Functions having identical names but different parameter lists are called *overloaded*. The act of defining such functions is called 'function overloading'.

The **C++** compiler implements function overloading in a rather simple way. Although the functions share their names (in this example `show`), the compiler (and hence the linker) use quite different names. The conversion of a name in the source file to an internally used name is called 'name mangling'. E.g.,

the **C++** compiler might convert the prototype `void show (int)` to the internal name `VshowI`, while an analogous function having a `char *` argument might be called `VshowCP`. The actual names that are used internally depend on the compiler and are not relevant for the programmer, except where these names show up in e.g., a listing of the contents of a library.

Some additional remarks with respect to function overloading:

- Do not use function overloading for functions doing conceptually different tasks. In the example above, the functions `show` are still somewhat related (they print information to the screen).

  However, it is also quite possible to define two functions `lookup`, one of which would find a name in a list while the other would determine the video mode. In this case the behavior of those two functions have nothing in common. It would therefore be more practical to use names which suggest their actions; say, `findname` and `videoMode`.

- **C++** does not allow identically named functions to differ only in their return values, as it is always the programmer's choice to either use or ignore a function's return value. E.g., the fragment

  ```
  printf("Hello World!\n");
  ```

  provides no information about the return value of the function `printf`. Two functions `printf` which only differ in their return types would therefore not be distinguishable to the compiler.

- In chapter 7 the notion of `const` member functions is introduced (cf. section 7.5). Here it is merely mentioned that classes normally have so-called member functions associated with them (see, e.g., chapter 5 for an informal introduction to the concept). Apart from overloading member functions using different parameter lists, it is then also possible to overload member functions by their `const` attributes. In those cases, classes may have pairs of identically named member functions, having identical parameter lists. Then, these functions are overloaded by their `const` attribute. In such cases only one of these function must have the `const` attribute.

2.5.5 Default function arguments

In **C++** it is possible to provide 'default arguments' when defining a function. These arguments are supplied by the compiler when they are not specified by the programmer. For example:

```
#include <stdio.h>

void showstring(char *str = "Hello World!\n");

int main()
{
    showstring("Here's an explicit argument.\n");

    showstring();           // in fact this says:
                            // showstring("Hello World!\n");
}
```

The possibility to omit arguments in situations where default arguments are defined is just a nice touch: it is the compiler who supplies the lacking argument unless it is explicitly specified at the call. The code of the program will neither be shorter nor more efficient when default arguments are used.

Functions may be defined with more than one default argument:

```
void two_ints(int a = 1, int b = 4);

int main()
{
```

```
    two_ints();             // arguments:  1, 4
    two_ints(20);           // arguments: 20, 4
    two_ints(20, 5);        // arguments: 20, 5
}
```

When the function `two_ints` is called, the compiler supplies one or two arguments whenever necessary. A statement like `two_ints(,6)` is, however, not allowed: when arguments are omitted they must be on the right-hand side.

Default arguments must be known at compile-time since at that moment arguments are supplied to functions. Therefore, the default arguments must be mentioned at the function's *declaration*, rather than at its *implementation*:

```
// sample header file
extern void two_ints(int a = 1, int b = 4);

// code of function in, say, two.cc
void two_ints(int a, int b)
{
    ...
}
```

It is an error to supply default arguments in function definitions. When the function is used by other sources the compiler reads the header file rather than the function definition. Consequently the compiler has no way to determine the values of default function arguments. Current compilers generate compile-time errors when detecting default arguments in function definitions.

2.5.6 NULL-pointers vs. 0-pointers and nullptr (C++0x)

In **C++** all zero values are coded as `0`. In **C** `NULL` is often used in the context of pointers. This difference is purely stylistic, though one that is widely adopted. In **C++** `NULL` should be avoided (as it is a macro, and macros can –and therefore should– easily be avoided in **C++**). Instead `0` can almost always be used.

Almost always, but not always. As **C++** allows function overloading (cf. section 2.5.4) the programmer might be confronted with an unexpected function selection in the situation shown in section 2.5.4:

```
#include <stdio.h>

void show(int val)
{
    printf("Integer: %d\n", val);
}

void show(double val)
{
    printf("Double: %lf\n", val);
}

void show(char const *val)
{
    printf("String: %s\n", val);
}

int main()
{
    show(12);
    show(3.1415);
```

```
    show("Hello World\n!");
}
```

In this situation a programmer intending to call `show(char const *)` might call `show(0)`. But this doesn't work, as 0 is interpreted as `int` and so `show(int)` is called. But calling `show(NULL)` doesn't work either, as **C++** usually defines `NULL` as 0, rather than `((void *)0)`. So, `show(int)` is called once again. To solve these kinds of problems the new **C++** standard introduces the keyword `nullptr` representing the 0 pointer. In the current example the programmer should call `show(nullptr)` to avoid the selection of the wrong function. The `nullptr` value can also be used to initialize pointer variables. E.g.,

```
int *ip = nullptr;      // OK
int value = nullptr;    // error: value is no pointer
```

The `nullptr` constant is not yet supported by the `g++` compiler.

2.5.7 The 'void' parameter list

In **C**, a function prototype with an empty parameter list, such as

```
void func();
```

means that the argument list of the declared function is not prototyped: the compiler will not warn against calling `func` with any set of arguments. In **C** the keyword `void` is used when it is the explicit intent to declare a function with no arguments at all, as in:

```
void func(void);
```

As **C++** enforces strict type checking, in **C++** an empty parameter list indicates the *total absence* of parameters. The keyword `void` is thus omitted.

2.5.8 The '#define __cplusplus'

Each **C++** compiler which conforms to the ANSI/ISO standard defines the symbol `__cplusplus`: it is as if each source file were prefixed with the preprocessor directive `#define __cplusplus`.

We shall see examples of the usage of this symbol in the following sections.

2.5.9 Using standard C functions

Normal **C** functions, e.g., which are compiled and collected in a run-time library, can also be used in **C++** programs. Such functions, however, must be declared as **C** functions.

As an example, the following code fragment declares a function `xmalloc` as a **C** function:

```
extern "C" void *xmalloc(int size);
```

This declaration is analogous to a declaration in **C**, except that the prototype is prefixed with `extern "C"`.

A slightly different way to declare **C** functions is the following:

```
extern "C"
```

```
{
    // C-declarations go in here
}
```

It is also possible to place preprocessor directives at the location of the declarations. E.g., a **C** header file `myheader.h` which declares **C** functions can be included in a **C++** source file as follows:

```
extern "C"
{
    #include <myheader.h>
}
```

Although these two approaches may be used, they are actually seldom encountered in **C++** sources. We will encounter a more frequently used method to declare external **C** functions in the next section.

2.5.10 Header files for both C and C++

The combination of the predefined symbol `__cplusplus` and the possibility to define `extern "C"` functions offers the ability to create header files for both **C** and **C++**. Such a header file might, e.g., declare a group of functions which are to be used in both **C** and **C++** programs.

The setup of such a header file is as follows:

```
#ifdef __cplusplus
extern "C"
{
#endif

    /* declaration of C-data and functions are inserted here. E.g., */
    void *xmalloc(int size);

#ifdef __cplusplus
}
#endif
```

Using this setup, a normal **C** header file is enclosed by `extern "C" {` which occurs near the top of the file and by `}`, which occurs near the bottom of the file. The `#ifdef` directives test for the type of the compilation: **C** or **C++**. The 'standard' **C** header files, such as `stdio.h`, are built in this manner and are therefore usable for both **C** and **C++**.

In addition **C++** headers should support *include guards*. In **C++** it is usually undesirable to include the same header file twice in the same source file. Such multiple inclusions can easily be avoided by including an `#ifndef` directive in the header file. For example:

```
#ifndef MYHEADER_H_
#define MYHEADER_H_
    // declarations of the header file is inserted here,
    // using #ifdef __cplusplus etc. directives
#endif
```

When this file is initially scanned by the preprocessor, the symbol `MYHEADER_H_` is not yet defined. The `#ifndef` condition succeeds and all declarations are scanned. In addition, the symbol `MYHEADER_H_` is defined.

When this file is scanned next while compiling the same source file, the symbol `MYHEADER_H_` has been defined and consequently all information between the `#ifndef` and `#endif` directives is skipped by the compiler.

In this context the symbol name `MYHEADER_H_` serves only for recognition purposes. E.g., the name of the header file can be used for this purpose, in capitals, with an underscore character instead of a dot.

Apart from all this, the custom has evolved to give **C** header files the extension `.h`, and to give `C++` header files *no* extension. For example, the standard *iostreams* `cin, cout` and `cerr` are available after including the header file `iostream`, rather than `iostream.h`. In the Annotations this convention is used with the standard **C++** header files, but not necessarily everywhere else.

There is more to be said about header files. Section 7.9 provides an in-depth discussion of the preferred organization of **C++** header files.

2.5.11 Defining local variables

In **C** local variables can only be defined at the top of a function or at the beginning of a nested block. In **C++** local variables can be created at any position in the code, even between statements.

Furthermore, local variables can be defined within some statements, just prior to their usage. A typical example is the `for` statement:

```
#include <stdio.h>

int main()
{
    for (int i = 0; i < 20; ++i)
        printf("%d\n", i);
}
```

In this program the variable `i` is created in the initialization section of the `for` statement. According to the ANSI-standard, the variable does not exist prior to the `for`-statement and not beyond the `for`-statement. With some older compilers, the variable continues to exist after the execution of the `for`-statement, but nowadays a warning like

> warning: name lookup of ‘i’ changed for new ANSI ‘for’ scoping using obsolete binding at ‘i’

is issued when the variable is used outside of the `for`-loop.

The implication seems clear: define a variable just before the `for`-statement if it is to be used beyond that statement. Otherwise the variable should be defined inside the `for`-statement itself. This reduces its scope as much as possible, which is a very desirable characteristic.

Defining local variables when they’re needed requires a little getting used to. However, eventually it tends to produce more readable, maintainable and often more efficient code than defining variables at the beginning of compound statements. We suggest the following rules of thumb for defining local variables:

- Local variables should be created at ‘intuitively right’ places, such as in the example above. This does not only entail the `for`-statement, but also all situations where a variable is only needed, say, half-way through the function.
- More in general, variables should be defined in such a way that their scope is as *limited* and *localized* as possible. When avoidable local variables are not defined at the beginning of functions but rather where they’re first used.
- It is considered good practice to *avoid global variables*. It is fairly easy to lose track of which global variable is used for what purpose. In **C++** global variables are seldom required, and by localizing variables the well known phenomenon of using the same variable for multiple purposes, thereby invalidating each individual purpose of the variable, can easily be prevented.

If considered appropriate, *nested blocks* can be used to localize auxiliary variables. However, situations exist where local variables are considered appropriate inside nested statements. The just mentioned `for` statement is of course a case in point, but local variables can also be defined within the condition clauses of `if-else` statements, within selection clauses of `switch` statements and condition clauses of `while` statements. Variables thus defined will be available to the full statement, including its nested statements. For example, consider the following `switch` statement:

```
#include <stdio.h>

int main()
{
    switch (int c = getchar())
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("Saw vowel %c\n", c);
        break;

        case EOF:
            printf("Saw EOF\n");
        break;

        default:
            printf("Saw other character, hex value 0x%2x\n", c);
    }
}
```

Note the location of the definition of the character 'c': it is defined in the expression part of the `switch` statement. This implies that 'c' is available *only* to the `switch` statement itself, including its nested (sub)statements, but not outside the scope of the `switch`.

The same approach can be used with `if` and `while` statements: a variable that is defined in the condition part of an `if` and `while` statement is available in their nested statements. There are some caveats, though:

- The variable definition must result in a variable which is initialized to a numeric or logical value;
- The variable definition cannot be nested (e.g., using parentheses) within a more complex expression.

The latter point of attention should come as no big surprise: in order to be able to evaluate the logical condition of an `if` or `while` statement, the value of the variable must be interpretable as either zero (false) or non-zero (true). Usually this is no problem, but in **C++** *objects* (like objects of the type `std::string` (cf. chapter 5)) are often returned by functions. Such objects may or may not be interpretable as numeric values. If not (as is the case with `std::string` objects), then such variables can *not* be defined at the condition or expression clauses of condition- or repetition statements. The following example will therefore *not* compile:

```
if (std::string myString = getString())     // assume getString returns
{                                           // a std::string value
    // process myString
}
```

The above example requires additional clarification. Often a variable can profitably be given local scope, but an extra check is required immediately following its initialization. The initialization *and* the test

cannot *both* be combined in one expression. Instead *two* nested statements are required. Consequently, the following example won't compile either:

```
if ((int c = getchar()) && strchr("aeiou", c))
    printf("Saw a vowel\n");
```

If such a situation occurs, either use two nested `if` statements, or localize the definition of `int c` using a nested compound statement:

```
if (int c = getchar())              // nested if-statements
    if (strchr("aeiou", c))
        printf("Saw a vowel\n");

{                                   // nested compound statement
    int c = getchar();
    if (c && strchr("aeiou", c))
       printf("Saw a vowel\n");
}
```

2.5.12 The keyword 'typedef'

The keyword `typedef` is still allowed in **C++**, but is not required anymore when defining `union`, `struct` or `enum` definitions. This is illustrated in the following example:

```
struct somestruct
{
    int     a;
    double  d;
    char    string[80];
};
```

When a `struct`, `union` or other compound type is defined, the tag of this type can be used as type name (this is `somestruct` in the above example):

```
somestruct what;

what.d = 3.1415;
```

2.5.13 Functions as part of a struct

In **C++** we may define functions as members of structs. Here we encounter the first concrete example of an object: as previously described (see section 2.4), an object is a structure containing data while specialized functions exist to manipulate those data.

A definition of a `struct Point` is provided by the code fragment below. In this structure, two `int` data fields and one function `draw` are declared.

```
struct Point            // definition of a screen-dot
{
    int x;              // coordinates
    int y;              // x/y
    void draw();        // drawing function
};
```

A similar structure could be part of a painting program and could, e.g., represent a pixel. With respect to this `struct` it should be noted that:

- The function `draw` mentioned in the `struct` definition is a mere *declaration*. The actual code of the function defining the actions performed by the function is found elsewhere (the concept of functions inside `structs` is further discussed in section 3.2).
- The size of the `struct Point` is equal to the size of its two `ints`. A function declared inside the structure does not affect its size. The compiler implements this behavior by allowing the function `draw` to be available only in the context of a `Point`.

The `Point` structure could be used as follows:

```
Point a;                // two points on
Point b;                // the screen

a.x = 0;                // define first dot
a.y = 10;               // and draw it
a.draw();

b = a;                  // copy a to b
b.y = 20;               // redefine y-coord
b.draw();               // and draw it
```

As shown in the above example a function that is part of the structure may be selected using the dot (.) (the arrow (`->`) operator is used when pointers to objects are available). This is therefore identical to the way data fields of structures are selected.

The idea behind this syntactic construction is that several types may contain functions having identical names. E.g., a structure representing a circle might contain three `int` values: two values for the coordinates of the center of the circle and one value for the radius. Analogously to the `Point` structure, a `Circle` may now have a function `draw` to draw the circle.

Chapter 3

A First Impression Of C++

In this chapter **C++** is further explored. The possibility to declare functions in `structs` is illustrated in various examples; the concept of a `class` is introduced; casting is covered in detail; many new types are introduced and several important notational extensions to **C** are discussed.

3.1 Extensions to C

Before we continue with the 'real' object-approach to programming, we first introduce some extensions to the **C** programming language: not mere differences between **C** and **C++**, but syntactic constructs and keywords not found in **C**.

3.1.1 Namespaces

C++ introduces the notion of a *namespace*: all symbols are defined in a larger context, called a *namespace*. Namespaces are used to avoid name conflicts that could arise when a programmer would like to define a function like `sin` operating on *degrees*, but does not want to lose the capability of using the standard `sin` function, operating on *radians*.

Namespaces are covered extensively in chapter 4. For now it should be noted that most compilers require the explicit declaration of a *standard namespace*: `std`. So, unless otherwise indicated, it is stressed that all examples in the Annotations now implicitly use the

```
using namespace std;
```

declaration. So, if you actually intend to compile examples given in the **C++** Annotations, make sure that the sources start with the above `using` declaration.

3.1.2 The scope resolution operator ::

C++ introduces several new operators, among which the scope resolution operator (`::`). This operator can be used in situations where a global variable exists having the same name as a local variable:

```
#include <stdio.h>

int counter = 50;                   // global variable

int main()
```

```
    {
        for (int counter = 1;           // this refers to the
             counter < 10;              // local variable
             counter++)
        {
            printf("%d\n",
                    ::counter           // global variable
                    /                   // divided by
                    counter);           // local variable
        }
    }
```

In the above program the scope operator is used to address a global variable instead of the local variable having the same name. In **C++** the scope operator is used extensively, but it is seldom used to reach a global variable shadowed by an identically named local variable. Its main purpose is described in chapter 7.

3.1.3 Using the keyword 'const'

Even though the keyword `const` is part of the **C** grammar, its use is more important and much more common in **C++** than it is in **C**.

The `const` keyword is a modifier stating that the value of a variable or of an argument may not be modified. In the following example the intent is to change the value of a variable `ival`, which fails:

```
int main()
{
    int const ival = 3;     // a constant int
                            // initialized to 3

    ival = 4;               // assignment produces
                            // an error message
}
```

This example shows how `ival` may be initialized to a given value in its definition; attempts to change the value later (in an assignment) are not permitted.

Variables that are declared `const` can, in contrast to **C**, be used to specify the size of an array, as in the following example:

```
int const size = 20;
char buf[size];             // 20 chars big
```

Another use of the keyword `const` is seen in the declaration of pointers, e.g., in pointer-arguments. In the declaration

```
char const *buf;
```

`buf` is a pointer variable pointing to `char`s. Whatever is pointed to by `buf` may not be changed through `buf`: the `char`s are declared as `const`. The pointer `buf` itself however may be changed. A statement like `*buf = 'a';` is therefore not allowed, while `++buf` is.

In the declaration

```
char *const buf;
```

`buf` itself is a `const` pointer which may not be changed. Whatever `chars` are pointed to by `buf` may be changed at will.

Finally, the declaration

```
char const *const buf;
```

is also possible; here, neither the pointer nor what it points to may be changed.

The rule of thumb for the placement of the keyword `const` is the following: whatever occurs to the *left* to the keyword may not be changed.

Although simple, this rule of thumb is often used. For example, Bjarne Stroustrup states (in `http://www.research.att.c`

> *Should I put "const" before or after the type?*
> *I put it before, but that's a matter of taste. "const T" and "T const" were always (both) allowed and equivalent. For example:*
>
> ```
> const int a = 1;        // OK
> int const b = 2;        // also OK
> ```
>
> *My guess is that using the first version will confuse fewer programmers ("is more idiomatic").*

But we've already seen an example where applying this simple 'before' placement rule for the keyword `const` produces unexpected (i.e., unwanted) results as we will shortly see below. Furthermore, the 'idiomatic' before-placement also conflicts with the notion of *const functions*, which we will encounter in section 7.5. With const functions the keyword `const` is also placed behind rather than before the name of the function.

The definition or declaration (either or not containing `const`) should always be read from the variable or function identifier back to the type indentifier:

> "Buf is a const pointer to const characters"

This rule of thumb is especially useful in cases where confusion may occur. In examples of **C++** code published in other places one often encounters the reverse: `const` *preceding* what should not be altered. That this may result in sloppy code is indicated by our second example above:

```
char const *buf;
```

What must remain constant here? According to the sloppy interpretation, the pointer cannot be altered (as `const` precedes the pointer). In fact, the char values are the constant entities here, as becomes clear when we try to compile the following program:

```
int main()
{
    char const *buf = "hello";

    ++buf;                  // accepted by the compiler
    *buf = 'u';             // rejected by the compiler
}
```

Compilation fails on the statement `*buf = 'u';` and *not* on the statement `++buf`.

Marshall Cline's `C++ FAQ`[1] gives the same rule (paragraph 18.5) , in a similar context:

> *[18.5] What's the difference between "const Fred* p", "Fred* const p" and "const Fred* const p"?*
> *You have to read pointer declarations right-to-left.*

[1] http://www.parashift.com/c++-faq-lite/const-correctness.html

Marshal Cline's advice might be improved, though: you should start to read pointer definitions (and declarations) at the variable name, reading as far as possible to the definition's end. Once you see a closing parenthesis, read backwards (right to left) from the initial point, until you find matching open-parenthesis or the very beginning of the definition. For example, consider the following complex declaration:

```
char const *(* const (*ip)[])[]
```

Here, we see:

- the variable `ip`, being a
- (reading backwards) modifiable pointer to an
- (reading forward) array of
- (reading backward) constant pointers to an
- (reading forward) array of
- (reading backward) modifiable pointers to constant characters

3.1.4 'cout', 'cin', and 'cerr'

Analogous to **C**, **C++** defines standard input- and output streams which are available when a program is executed. The streams are:

- `cout`, analogous to `stdout`,
- `cin`, analogous to `stdin`,
- `cerr`, analogous to `stderr`.

Syntactically these streams are not used as functions: instead, data are written to streams or read from them using the operators $<<$, called the *insertion operator* and $>>$, called the *extraction operator*. This is illustrated in the next example:

```
#include <iostream>

using namespace std;

int main()
{
    int     ival;
    char    sval[30];

    cout << "Enter a number:\n";
    cin >> ival;
    cout << "And now a string:\n";
    cin >> sval;

    cout << "The number is: " << ival << "\n"
            "And the string is: " << sval << '\n';
}
```

This program reads a number and a string from the `cin` stream (usually the keyboard) and prints these data to `cout`. With respect to streams, please note:

- The standard streams are declared in the header file `iostream`. In the examples in the **C++** Annotations this header file is often not mentioned explicitly. Nonetheless, it *must* be included

(either directly or indirectly) when these streams are used. Comparable to the use of the `using namespace std;` clause, the reader is expected to `#include <iostream>` with all the examples in which the standard streams are used.

- The streams `cout`, `cin` and `cerr` are variables of so-called *class*-types. Such variables are commonly called *objects*. Classes are discussed in detail in chapter 7 and are used extensively in **C++**.
- The stream `cin` extracts data from a stream and copies the extracted information to variables (e.g., `ival` in the above example) using the extraction operator (two consecutive `>` characters: `>>`). We will describe later how operators in **C++** can perform quite different actions than what they are defined to do by the language, as is the case here. Function overloading has already been mentioned. In **C++** *operators* can also have multiple definitions, which is called *operator overloading*.
- The operators which manipulate `cin`, `cout` and `cerr` (i.e., `>>` and `<<`) also manipulate variables of different types. In the above example `cout << ival` results in the printing of an integer value, whereas `cout << "Enter a number"` results in the printing of a string. The actions of the operators therefore depend on the types of supplied variables.
- The *extraction operator* (`>>`) performs a so called *type safe* assignment to a variable by 'extracting' its value from a text stream. Normally, the extraction operator skips all *white space* characters preceding the values to be extracted.
- Special symbolic constants are used for special situations. Normally a line is terminated by inserting `"\n"` or `'\n'`. But when inserting the `endl` symbol the line is terminated followed by the flushing of the stream's internal buffer. Thus, `endl` can usually be avoided in favor of `'\n'` resulting in somewhat more efficient code.

The stream objects `cin`, `cout` and `cerr` are not part of the **C++** grammar proper. The streams are part of the definitions in the header file `iostream`. This is comparable to functions like `printf` that are not part of the **C** grammar, but were originally written by people who considered such functions important and collected them in a run-time library.

A program may still use the old-style functions like `printf` and `scanf` rather than the new-style streams. The two styles can even be mixed. But streams offer several clear advantages and in many **C++** programs have completely replaced the old-style **C** functions. Some advantages of using streams are:

- Using insertion and extraction operators is *type-safe*. The format strings which are used with `printf` and `scanf` can define wrong format specifiers for their arguments, for which the compiler sometimes can't warn. In contrast, argument checking with `cin`, `cout` and `cerr` is performed by the compiler. Consequently it isn't possible to err by providing an `int` argument in places where, according to the format string, a string argument should appear. With streams there are no format strings.
- The functions `printf` and `scanf` (and other functions using format strings) in fact implement a *mini-language* which is interpreted run-time. In contrast, with streams the **C++** compiler knows exactly which in- or output action to perform given the arguments used. No mini-language here.
- In addition the possibilities of the insertion and extraction operators may be *extended* allowing objects of classes that didn't exist when the streams were originally designed to be inserted into or extracted from streams. Mini languages as used with `printf` cannot be extended.
- The usage of the left-shift and right-shift operators in the context of the streams illustrates yet another capability of **C++**: operator overloading allowing us to redefine the actions an operator performs in certain contexts. Ascending from **C** operator overloading requires some getting used, but after a short little while these overloaded operators feel rather comfortable.
- Streams are independent of the media they operate upon. This (at this point somewhat abstract) notion means that the same code can be used without *any* modification at all to interface your code to *any* kind of device. The code using streams can be used when the device is a file on disk; an

Internet connection; a digital camera; a DVD device; a satellite link; and much more: you name it. Streams allow your code to be decoupled (independent) of the devices your code is supposed to operate on, which eases maintenance and allows reuse of the same code in new situations.

The *iostream library* has a lot more to offer than just `cin`, `cout` and `cerr`. In chapter 6 *iostreams* will be covered in greater detail. Even though `printf` and friends can still be used in **C++** programs, streams have practically replaced the old-style **C** `I/O` functions like `printf`. If you *think* you still need to use `printf` and related functions, think again: in that case you've probably not yet completely grasped the possibilities of stream objects.

3.2 Functions as part of structs

Earlier it was mentioned that functions can be part of `structs` (see section 2.5.13). Such functions are called *member functions*. This section briefly discusses how to define such functions.

The code fragment below shows a `struct` having data fields for a person's name and address. A function `print` is included in the `struct`'s definition:

```
struct Person
{
    char name[80];
    char address[80];

    void print();
};
```

When defining the member function `print` the structure's name (`Person`) and the scope resolution operator (`::`) are used:

```
void Person::print()
{
    cout << "Name:      " << name << "\n"
            "Address:   " << address << '\n';
}
```

The implementation of `Person::print` shows how the fields of the `struct` can be accessed without using the structure's type name. Here the function `Person::print` prints a variable `name`. Since `Person::print` is itself a part of `struct person`, the variable `name` implicitly refers to the same type.

This `struct Person` could be used as follows:

```
Person person;

strcpy(person.name, "Karel");
strcpy(p.address, "Marskramerstraat 33");
p.print();
```

The advantage of member functions is that the called function automatically accesses the data fields of the structure for which it was invoked. In the statement `person.print()` the object `person` is the 'substrate': the variables `name` and `address` that are used in the code of `print` refer to the data stored in the `person` object.

3.2.1 Data hiding: public, private and class

As mentioned before (see section 2.3), **C++** contains specialized syntactic possibilities to implement data hiding. Data hiding is the capability of sections of a program to hide its data from other sections. This results in very clean data definitions. It also allows these sections to enforce the integrity of their data.

C++ has three keywords that are related to data hiding: `private`, `protected` and `public`. These keywords can be used in the definition of `structs`. The keyword `public` allows all subsequent fields of a structure to be accessed by all code; the keyword `private` only allows code that is part of the `struct` itself to access subsequent fields. The keyword `protected` is discussed in chapter 13, and is somewhat outside of the scope of the current discussion.

In a `struct` all fields are `public`, unless explicitly stated otherwise. Using this knowledge we can expand the `struct Person`:

```
struct Person
{
    private:
        char d_name[80];
        char d_address[80];
    public:
        void setName(char const *n);
        void setAddress(char const *a);
        void print();
        char const *name();
        char const *address();
};
```

As the data fields `d_name` and `d_address` are in a `private` section they are only accessible to the member functions which are defined in the `struct`: these are the functions `setName`, `setAddress` etc.. As an illustration consider the following code:

```
Person fbb;

fbb.setName("Frank");         // OK, setName is public
strcpy(fbb.d_name, "Knarf");  // error, x.d_name is private
```

Data integrity is implemented as follows: the actual data of a `struct Person` are mentioned in the structure definition. The data are accessed by the outside world using special functions that are also part of the definition. These member functions control all traffic between the data fields and other parts of the program and are therefore also called 'interface' functions. The thus implemented data hiding is illustrated in Figure 3.1. The members `setName` and `setAddress` are declared with `char const *` parameters. This indicates that the functions will not alter the strings which are supplied as their arguments. Analogously, the members `name` and `address` return `char const *`s: the compiler will prevent callers of those members from modifying the information made accessible through the return values of those members.

Two examples of member functions of the `struct Person` are shown below:

```
void Person::setName(char const *n)
{
    strncpy(d_name, n, 79);
    d_name[79] = 0;
}

char const *Person::name()
{
```

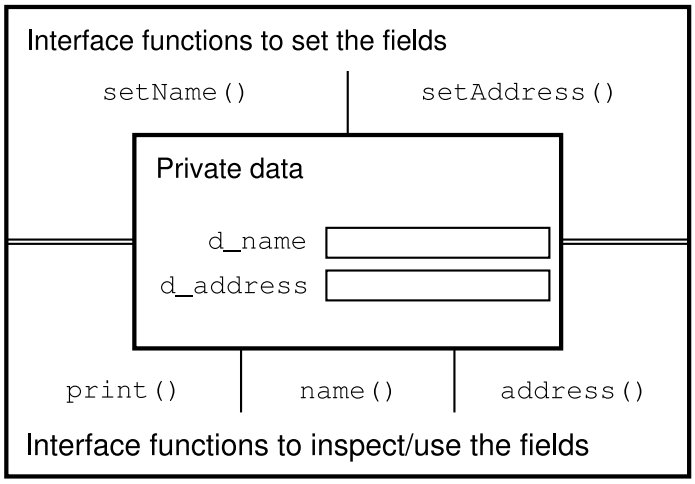

Figure 3.1: Private data and public interface functions of the class Person.

```
    return d_name;
}
```

The power of member functions and of the concept of data hiding results from the abilities of member functions to perform special tasks, e.g., checking the validity of the data. In the above example `setName` copies only up to 79 characters from its argument to the data member `name`, thereby avoiding a buffer overflow.

Another illustration of the concept of data hiding is the following. As an alternative to member functions that keep their data in memory a library could be developed featuring member functions storing data on file. To convert a program which stores `Person` structures in memory to one that stores the data on disk no special modifications would be required. After recompilation and linking the program to a new library it will have converted from storage in memory to storage on disk. This example illustrates a broader concept than data hiding; it illustrates *encapsulation*. Data hiding is a kind of encapsulation. Encapsulation in general results in reduced coupling of different sections of a program. This in turn greatly enhances reusability and maintainability of the resulting software. By having the structure encapsulate the actual storage medium the program using the structure becomes independent of the actual storage medium that is used.

Though data hiding can be implemented using `structs`, more often (almost always) *classes* are used instead. A class is a kind of struct, except that a class uses private access by default, whereas structs use public access by default. The definition of a `class Person` is therefore identical to the one shown above, except for the fact that the keyword `class` has replaced `struct` while the initial `private:` clause can be omitted. Our typographic suggestion for class names (and other type names defined by the programmer) is to start with a capital character to be followed by the remainder of the type name using lower case letters (e.g., `Person`).

3.2.2 Structs in C vs. structs in C++

In this section we'll discuss an important difference between **C** and **C++** structs and (member) functions. In **C** it is common to define several functions to process a `struct`, which then require a pointer to the `struct` as one of their arguments. An imaginary **C** header file showing this concept is:

```
/* definition of a struct PERSON    This is C   */
typedef struct
{
    char name[80];
    char address[80];
} PERSON;
```

```
/* some functions to manipulate PERSON structs */

/* initialize fields with a name and address    */
void initialize(PERSON *p, char const *nm,
                  char const *adr);

/* print information    */
void print(PERSON const *p);

/* etc..    */
```

In **C++**, the declarations of the involved functions are put inside the definition of the `struct` or `class`. The argument denoting which `struct` is involved is no longer needed.

```
class Person
{
    char d_name[80];
    char d_address[80];

    public:
        void initialize(char const *nm, char const *adr);
        void print();
        // etc..
};
```

In **C++** the `struct` parameter is not used. A **C** function call such as:

```
PERSON x;

initialize(&x, "some name", "some address");
```

becomes in **C++**:

```
Person x;

x.initialize("some name", "some address");
```

3.3 More extensions to C

3.3.1 References

In addition to the common ways to define variables (plain variables or pointers) **C++** introduces *references* defining synonyms for variables. A reference to a variable is like an *alias*; the variable and the reference can both be used in statements involving the variable:

```
int int_value;
int &ref = int_value;
```

In the above example a variable `int_value` is defined. Subsequently a reference `ref` is defined, which (due to its initialization) refers to the same memory location as `int_value`. In the definition of `ref`, the reference operator & indicates that `ref` is not itself an `int` but a reference to one. The two statements

```
++int_value;
++ref;
```

have the same effect: they increment `int_value`'s value. Whether that location is called `int_value` or `ref` does not matter.

References serve an important function in **C++** as a means to pass modifiable arguments to functions. E.g., in standard **C**, a function that increases the value of its argument by five and returning nothing needs a pointer parameter:

```
void increase(int *valp)    // expects a pointer
{                           // to an int
    *valp += 5;
}

int main()
{
    int x;

    increase(&x);           // pass x's address
}
```

This construction can *also* be used in **C++** but the same effect is also achieved using a reference:

```
void increase(int &valr)    // expects a reference
{                           // to an int
    valr += 5;
}

int main()
{
    int x;

    increase(x);            // passed as reference
}
```

It is arguable whether code such as the above should be preferred over **C**'s method, though. The statement `increase (x)` suggests that not `x` itself but a *copy* is passed. Yet the value of `x` changes because of the way `increase()` is defined. However, references can also be used to pass objects that are only inspected (without the need for a copy or a const *) or to pass objects whose modification is an accepted side-effect of their use. In those cases using references are strongly preferred over existing alternatives like copy by value or passing pointers.

Behind the scenes references are implemented using pointers. So, as far as the compiler is concerned references in **C++** are just const pointers. With references, however, the programmer does not need to know or to bother about levels of indirection. An important distinction between plain pointers and references is of course that with references no indirection takes place. For example:

```
extern int *ip;
extern int &ir;

ip = 0;     // reassigns ip, now a 0-pointer
ir = 0;     // ir unchanged, the int variable it refers to
            // is now 0.
```

In order to prevent confusion, we suggest to adhere to the following:

- In those situations where a function does not alter its parameters of a built-in or pointer type, value parameters can be used:

  ```
  void some_func(int val)
  ```

```
{
    cout << val << endl;
}

int main()
{
    int x;

    some_func(x);       // a copy is passed
}
```

- When a function explicitly must change the values of its arguments, a pointer parameter is preferred. These pointer parameters should preferably be the function's initial parameters. This is called *return by argument*.

```
void by_pointer(int *valp)
{
    *valp += 5;
}
```

- When a function doesn't change the value of its class- or struct-type arguments, or if the modification of the argument is a trivial side-effect (e.g., the argument is a stream) references can be used. Const-references should be used if the function does not modify the argument:

```
void by_reference(string const &str)
{
    cout << str;    // no modification of str
}

int main ()
{
    int x = 7;
    by_pointer(&x);         // a pointer is passed
                            // x might be changed
    string str("hello");
    by_reference(str);      // str is not altered
}
```

References play an important role in cases where the argument is not changed by the function but where it is undesirable to copy the argument to initialize the parameter. Such a situation occurs when a large object is passed as argument, or is returned by the function. In these cases the copying operation tends to become a significant factor, as the entire object must be copied. In these cases references are preferred.

If the argument isn't modified by the function, or if the caller shouldn't modify the returned information, the `const` keyword should be used. Consider the following example:

```
struct Person                   // some large structure
{
    char    name[80];
    char    address[90];
    double  salary;
};

Person person[50];          // database of persons

                            // printperson expects a
                            // reference to a structure
                            // but won't change it
void printperson (Person const &p)
```

```
{
    cout << "Name: " << p.name << endl <<
            "Address: " << p.address << endl;

}
                            // get a person by indexvalue
Person const &person(int index)
{
    return person[index];   // a reference is returned,
}                           // not a copy of person[index]

int main()
{
    Person boss;

    printperson (boss);     // no pointer is passed,
                            // so variable won't be
                            // altered by the function
    printperson(person(5));
                            // references, not copies
                            // are passed here
}
```

- Furthermore, note that there is yet another reason for using references when passing objects as function arguments. When passing a reference to an object, the activation of a so called *copy constructor* is avoided. Copy constructors will be covered in chapter 8.

References *could* result in extremely 'ugly' code. A function may return a reference to a variable, as in the following example:

```
int &func()
{
    static int value;
    return value;
}
```

This allows the use of the following constructions:

```
func() = 20;
func() += func();
```

It is probably superfluous to note that such constructions should normally not be used. Nonetheless, there are situations where it is useful to return a reference. We have actually already seen an example of this phenomenon in our previous discussion of streams. In a statement like `cout << "Hello" << '\n';` the insertion operator returns a reference to `cout`. So, in this statement first the `"Hello"` is inserted into `cout`, producing a reference to `cout`. Through this reference the `'\n'` is then inserted in the `cout` object, again producing a reference to `cout`, which is then ignored.

Several differences between pointers and references are pointed out in the next list below:

- A reference cannot exist by itself, i.e., without something to refer to. A declaration of a reference like

  ```
  int &ref;
  ```

  is not allowed; what would `ref` refer to?

- References can be declared as `external`. These references were initialized elsewhere.

- References may exist as parameters of functions: they are initialized when the function is called.
- References may be used in the return types of functions. In those cases the function determines what the return value will refer to.
- References may be used as data members of classes. We will return to this usage later.
- Pointers are variables by themselves. They point at something concrete or just “at nothing”.
- References are aliases for other variables and cannot be re-aliased to another variable. Once a reference is defined, it refers to its particular variable.
- Pointers (except for const pointers) can be reassigned to point to different variables.
- When an address-of operator & is used with a reference, the expression yields the address of the variable to which the reference applies. In contrast, ordinary pointers are variables themselves, so the address of a pointer variable has nothing to do with the address of the variable pointed to.

3.3.2 Rvalue References (C++0x)

In **C++**, temporary (rvalue) values are indistinguishable from `const &` types. the C++0x standard adds a new reference type called an *rvalue reference*, defined as `typename &&`.

The name *rvalue* reference is derived from assignment statements, where the variable to the left of the assignment operator is called an *lvalue* and the expression to the right of the assignment operator is called an *rvalue*. Rvalues are often temporary (or anonymous) values, like values returned by functions.

In this parlance the **C++** reference should be considered an *lvalue reference* (using the notation `typename &`). They can be contrasted to *rvalue references* (using the notation `typename &&`).

The key to understanding rvalue references is *anonymous variable*. An anonymous variable has no name and this is the distinguishing feature for the compiler to associate it automatically with an lvalue reference if it has a choice. Before introducing some interesting and new constructions that weren’t available before C++0x let’s first have a look at some distinguishing applications of lvalue references. The following function returns a temporary (anonymous) value:

```
int intVal()
{
    return 5;
}
```

Although the return value of `intVal` can be assigned to an `int` variable it requires a copying operation, which might become prohibitive when a function does not return an `int` but instead some large object. A *reference* or *pointer* cannot be used either to collect the anonymous return value as the return value won’t survive beyond that. So the following is illegal (as noted by the compiler):

```
int &ir = intVal();         // fails: refers to a temporary
int const &ic = intVal();   // OK: immutable temporary
int *ip = &intVal();        // fails: no lvalue available
```

Apparently it is not possible to modify the temporary returned by `intVal`. But now consider the next function:

```
void receive(int &value)
{
    cout << "int value parameter\n";
}
void receive(int &&value)
{
    cout << "int R-value parameter\n";
}
```

and let's call this function from `main`:

```
int main()
{
    receive(18);
    int value = 5;
    receive(value);
    receive(intVal());
}
```

This program produces the following output:

```
int R-value parameter
int value parameter
int R-value parameter
```

It shows the compiler selecting `receive(int &&value)` in all cases where it receives an anonymous `int` as its argument. Note that this includes `receive(18)`: a value 18 has no name and thus `receive(int &&value)` is called. Internally, it actually uses a temporary variable to store the 18, as is shown by the following example which modifies `receive`:

```
void receive(int &&value)
{
    ++value;
    cout << "int R-value parameter, now: " << value << '\n';
        // displays 19 and 6, respectively.
}
```

Contrasting `receive(int &value)` with `receive(int &&value)` has nothing to do with `int &value` not being a const reference. If `receive(int const &value)` is used the same results are obtained. Bottom line: the compiler selects the overloaded function using the rvalue reference if the function is passed an anonymous value.

The compiler runs into problems if `void receive(int &value)` is replaced by `void receive(int value)`, though. When confronted with the choice between a value parameter and a reference parameter (either lvalue or rvalue) it cannot make a decision and reports an ambiguity. In practical contexts this is not a problem. Rvalue refences were added to the language in order to be able to distinguish the two forms of references: named values (for which lvalue references are used) and anonymous values (for which rvalue references are used).

It is this distinction that allows the implementation of *move semantics* and *perfect forwarding*. At this point the concept of *move semantics* cannot yet fully be discussed (but see section 8.6 for a more thorough discussusion) but it is very well possible to illustrate the underlying ideas.

Consider the situation where a function returns a `struct Data` containing a pointer to dynamically allocated characters. Moreover, the struct defines a member function `copy(Data const &other)` that takes another `Data` object and copies the other's data into the current object. The (partial) definition of the `struct Data` might look like this[2]:

```
struct Data
{
    char *text;
    size_t size;
    void copy(Data const &other)
    {
        text = strdup(other.text);
```

[2] To the observant reader: in this example the memory leak that results from using Data::copy() should be ignored

```
            size = strlen(text);
        }
    };
```

Next, functions `dataFactory` and `main` are defined as follows:

```
    Data dataFactory(char const *txt)
    {
        Data ret = {strdup(txt), strlen(txt)};
        return ret;
    }

    int main()
    {
        Data d1 = {strdup("hello"), strlen("hello")};

        Data d2;
        d2.copy(d1);                        // 1 (see text)

        Data d3;
        d3.copy(dataFactory("hello"));      // 2
    }
```

At (1) `d2` appropriately receives a copy of `d1`'s text. But at (2) `d3` receives a copy of the text stored in the temporary returned by the `dataFactory` function. As the temporary ceases to exist after the call to `copy()` two releated and unpleasant consequences are observed:

- The return value is a temporary object: its only reason for existence is to pass its data on to `d3`. Now `d3` copies the temporary's data which clearly is somewhat overdone.
- The temporary `Data` object is lost following the call to `copy()`. Unfortunately its dynamically allocated data is lost as well resulting in a memory leak.

In cases like these *rvalue reference* should be used. By overloading the `copy` member with a member `copy(Data &&other)` the compiler is able to distinguish situations (1) and (2). It now calls the initial `copy()` member in situation (1) and the newly defined overloaded `copy()` member in situation (2):

```
    struct Data
    {
        char *text;
        size_t size;
        void copy(Data const &other)
        {
            text = strdup(other.text);
        }
        void copy(Data &&other)
        {
            text = other.text;
            other.text = 0;
        }
    };
```

Note that the overloaded `copy()` function merely moves the `other.text` pointer to the current object's `text` pointer followed by reassigning 0 to `other.text`. `Struct Data` suddenly has become *move-aware* and implements *move semantics*, removing the drawbacks of the previously shown approach:

- Instead of making a deep copy (which is required in situation (1)), the pointer value is simply moved to its new owner;

- Since the `other.text` doesn't point to dynamically allocated memory anymore the memory leak is prevented.

Rvalue references for `*this` and initialization of class objects by rvalues are not yet supported by the `g++` compiler.

3.3.3 Strongly typed enumerations (C++0x, 4.4)

Enumeration values in **C++** are in fact `int` values, thereby bypassing type safety. E.g., values of different enumeration types may be compared for (in)equality, albeit through a (static) type cast.

Another problem with the current `enum` type is that their values are not restricted to the enum type name itself, but to the scope where the enumeration is defined. As a consequence, two enumerations having the same scope cannot have identical values.

In the C++0x standard these problems are solved by defining *enum classes*. An *enum class* can be defined as in the following example:

```
enum class SafeEnum
{
    NOT_OK,     // 0, by implication
    OK          = 10,
    MAYBE_OK    // 11, by implication
};
```

Enum classes use `int` values by default, but the used value type can easily be changed using the `: type` notation, as in:

```
enum class CharEnum: unsigned char
{
    NOT_OK,
    OK
};
```

To use a value defined in an enum class its enumeration name must be provided as well. E.g., `OK` is not defined, `CharEnum::OK` is.

Using the data type specification (noting that it defaults to `int`) it is possible to use enum class forward declarations. E.g.,

```
enum Enum1;                 // Illegal: no size available
enum Enum2: unsigned int;   // Legal in C++0x: explicitly declared type

enum class Enum3;           // Legal in C++0x: default int type is used
enum class Enum4: char;     // Legal in C++0x: explicitly declared type
```

3.3.4 Initializer lists (C++0x, 4.4)

The **C** language defines the initializer list as a list of values enclosed by curly braces, possibly themselves containing initializer lists. In **C** these initializer lists are commonly used to initialize arrays and structs.

C++ extends this concept in the C++0x standard by introducing the *type* `initializer_list<Type>` where `Type` is replaced by the type name of the values used in the initializer list. Initializer lists in **C++** are, like their counterparts in **C**, recursive, so they can also be used with multi-dimensional arrays, structs and classes.

Like in **C**, initializer lists consist of a list of values surrounded by curly braces. But unlike **C**, *functions* can define initializer list parameters. E.g.,

```
void values(std::initializer_list<int> iniValues)
{
}
```

A function like `arrayValues` could be called as follows:

```
values({2, 3, 5, 7, 11, 13});
```

The initializer list appears as an argument which is a list of values surrounded by curly braces. Due to the recursive nature of initializer lists a two-dimensional series of values can also be passes, as shown in the next example:

```
void values2(std::initializer_list<int> iniValues)
{
}

values2({{1, 2}, {2, 3}, {3, 5}, {4, 7}, {5, 11}, {6, 13}});
```

Initializer lists are constant expressions and cannot be modified. However, their *size* and values may be retrieved using their `size, begin`, and `end` members as follows:

```
void values(initializer_list<int> iniValues)
{
    cout << "Initializer list having " << iniValues.size() << "values\n";
    for
    (
        initializer_list<int>::const_iterator begin = iniValues.begin();
            begin != iniValues.end();
                ++begin
    )
        cout << "Value: " << *begin << '\n';
}
```

Initializer lists can also be used to initialize objects of classes (cf. section 7.3).

3.3.5 Type inference using 'auto' (C++0x, 4.4)

A special use of the keyword `auto` is defined by the C++0x standard allowing the compiler to determine the type of a variable automatically rather than requiring the software engineer to define a variable's type explicitly.

In parallel, the use of `auto` as a storage class specifier is no longer supported in the C++0x standard. According to that standard a variable definition like `auto int var` results in a compilation error.

This can be very useful in situations where it is very hard to determine the variable's type in advance. These situations occur, e.g., in the context of *templates*, topics covered in chapters 18 until 22.

At this point in the Annotations only simple examples can be given, and some hints will be provided about more general uses of the `auto` keyword.

When defining and initializing a variable `int variable = 5` the type of the initializing expression is well known: it's an `int`, and unless the programmer's intentions are different this could be used

to define variable's type (although it shouldn't in normal circumstances as it reduces rather than improves the clarity of the code):

```
auto variable = 5;
```

Here are some examples where using auto *is* useful. In chapter 5 the *iterator* concept is introduced (see also chapters 12 and 18). Iterators sometimes have long type definitions, like

```
std::vector<std::string>::const_reverse_iterator
```

Functions may return types like this. Since the compiler knows the types returned by functions we may exploit this knowledge using auto. Assuming that a function begin() is declared as follows:

```
std::vector<std::string>::const_reverse_iterator begin();
```

Rather than writing the verbose variable definition (at // 1) a much shorter definition (at // 2) may be used:

```
std::vector<std::string>::const_reverse_iterator iter = begin();    // 1
auto iter = begin();                                                // 2
```

It's easy to define additional variables of this type. When initializing those variables using iter the auto keyword can be used again:

```
auto start = iter;
```

If start can't be initialized immediately using an existing variable the type of a well known variable of function can be used in combination with the decltype keyword, as in:

```
decltype(iter) start;
decltype(begin()) spare;
```

The keyword decltype may also receive an expression as its argument. This feature is already available in the C++0x standard implementation in g++ 4.3. E.g., decltype(3 + 5) represents an int, decltype(3 / double(3)) represents double.

The auto keyword can also be used to postpone the definition of a function's return type. The declaration of a function intArrPtr returning a pointer to an array of 10 ints looks like this:

```
int (*intArrPtr())[10];
```

Such a declaration is fairly complex. E.g., among other complexities it requires 'protection of the pointer' using parentheses in combination with the function's parameter list. In situations like these the specification of the return type can be postponed using the auto return type, followed by the specification of the function's return type after any other specification the function might receive (e.g., as a const member (cf. section 7.5) or following its exception throw list (cf. section 9.6)).

Using auto to declare the above function, the declaration becomes:

```
auto intArrPtr() -> int (*)[10];
```

A return type specification using auto is called a *late-specified return type*.

3.3.6 Range-based for-loops (C++0x, ?)

The **C++** for-statement is identical to **C**'s for-statement:

```
for (init; cond; inc)
    statement
```

In many cases, however, the initialization, condition and increment parts are fairly obvious as in situations where all elements of an array or vector must be processed. Many languages offer the `foreach` statement for that and **C++** offers the `std::for_each` generic algorithm (cf. section 19.1.17).

The C++0x standard adds a new `for` statement syntax to this. The new syntax can be used to process each element of a range. Three types of ranges are distinguished:

- Plain arrays (e.g., `int array[10]`);
- Standard containers (or comparable) (cf. chapter 12);
- Ranges contained in a `std::pair` (cf. section 12.2, e.g., `std::pair subRange(array + 1, array + 8)`).

In these cases the C++0x standard offers the following additional for-statement syntax:

```
// assume int array[30]
for (int &element: array)
    statement
```

here an `int &element` is defined whose lifetime and scope is restricted to the lifetime of the for-statement. It refers to each of the subsequent elements of `array` at each new iteration of the for-statement, starting with the first element of the range.

3.4 New language-defined data types

In **C** the following built-in data types are available: `void, char, short, int, long, float` and `double`. **C++** extends these built-in types with several additional built-in types: the types `bool`, `wchar_t`, `long long` and `long double` (Cf. ANSI/ISO draft (1995), par. 27.6.2.4.1 for examples of these very long types). The type `long long` is merely a double-long `long` datatype. The type `long double` is merely a double-long `double` datatype. These built-in types as well as pointer variables are called *primitive types* in the **C++** Annotations.

Except for these built-in types the class-type `string` is available for handling character strings. The datatypes `bool`, and `wchar_t` are covered in the following sections, the datatype `string` is covered in chapter 5. Note that recent versions of **C** may also have adopted some of these newer data types (notably `bool` and `wchar_t`). Traditionally, however, **C** doesn't support them, hence they are mentioned here.

Now that these new types are introduced, let's refresh your memory about *letters* that can be used in *literal constants* of various types. They are:

- `E` or `e`: the *exponentiation* character in floating point literal values. For example: `1.23E+3`. Here, `E` should be pronounced (and interpreted) as: *times 10 to the power*. Therefore, `1.23E+3` represents the value `1230`.
- `F` can be used as *postfix* to a non-integral numeric constant to indicate a value of type `float`, rather than `double`, which is the default. For example: `12.F` (the dot transforms 12 into a floating point value); `1.23E+3F` (see the previous example. `1.23E+3` is a `double` value, whereas `1.23E+3F` is a `float` value).

- `L` can be used as *prefix* to indicate a character string whose elements are `wchar_t`-type characters. For example: `L"hello world"`.
- `L` can be used as *postfix* to an integral value to indicate a value of type `long`, rather than `int`, which is the default. Note that there is no letter indicating a `short` type. For that a `static_cast<short>()` must be used.
- `p`, to specify the power in hexadecimal floating point numbers. E.g. `0x10p4`. The exponent itself is read as a decimal constant and can therefore not start with 0x. The exponent part is interpreted as a power of 2. So `0x10p2` is (decimal) equal to 64: `16 * 2^2`.
- `U` can be used as *postfix* to an integral value to indicate an `unsigned` value, rather than an `int`. It may also be combined with the postfix `L` to produce an `unsigned long int` value.

And, of course: the `x` and `a` until `f` characters can be used to specify hexadecimal constants (optionally using capital letters).

3.4.1 The data type 'bool'

The type `bool` represents boolean (logical) values, for which the (now reserved) constants `true` and `false` may be used. Except for these reserved values, integral values may also be assigned to variables of type `bool`, which are then implicitly converted to `true` and `false` according to the following conversion rules (assume `intValue` is an `int`-variable, and `boolValue` is a `bool`-variable):

```
    // from int to bool:
boolValue = intValue ? true : false;

    // from bool to int:
intValue = boolValue ? 1 : 0;
```

Furthermore, when `bool` values are inserted into streams then `true` is represented by `1`, and `false` is represented by `0`. Consider the following example:

```
cout << "A true value: "  << true << "\n"
        "A false value: " << false << '\n';
```

The `bool` data type is found in other programming languages as well. **Pascal** has its type `Boolean`; **Java** has a `boolean` type. Different from these languages, **C++**'s type `bool` acts like a kind of `int` type. It is primarily a documentation-improving type, having just two values `true` and `false`. Actually, these values can be interpreted as `enum` values for `1` and `0`. Doing so would ignore the philosophy behind the `bool` data type, but nevertheless: assigning `true` to an `int` variable neither produces warnings nor errors.

Using the `bool`-type is usually clearer than using `int`. Consider the following prototypes:

```
bool exists(char const *fileName);  // (1)
int  exists(char const *fileName);  // (2)
```

For the first prototype, readers will expect the function to return `true` if the given filename is the name of an existing file. However, using the second prototype some ambiguity arises: intuitively the return value 1 is appealing, as it allows constructions like

```
if (exists("myfile"))
    cout << "myfile exists";
```

On the other hand, many system functions (like **access**(2), **stat**(2), and many other) return 0 to indicate a successful operation, reserving other values to indicate various types of errors.

As a rule of thumb I suggest the following: if a function should inform its caller about the success or failure of its task, let the function return a `bool` value. If the function should return success or various types of errors, let the function return *enum* values, documenting the situation by its various symbolic constants. Only when the function returns a conceptually meaningful integral value (like the sum of two `int` values), let the function return an `int` value.

3.4.2 The data type 'wchar_t'

The `wchar_t` type is an extension of the `char` built-in type, to accomodate *wide* character values (but see also the next section). The `g++` compiler reports `sizeof(wchar_t)` as 4, which easily accomodates all 65,536 different *Unicode* character values.

Note that **Java**'s `char` data type is somewhat comparable to **C++**'s `wchar_t` type. **Java**'s `char` type is 2 bytes wide, though. On the other hand, **Java**'s `byte` data type is comparable to **C++**'s `char` type: one byte. Confusing?

3.4.3 Unicode encoding (C++0x, 4.4)

In **C++** string literals can be defined as ASCII-Z **C** strings. Prepending an ASCII-Z string by `L` (e.g., `L"hello"`) defines a `wchar_t` string literal.

The new C++0x standard adds to this support for 8, 16 and 32 bit Unicode encoded strings. Furthermore, two new data types are introduced: `char16_t` and `char32_t` storing, respectively, a `UTF-16` and `UTF-32` unicode value.

In addition, `char` will be large enough to contain any `UTF-8` unicode value as well (i.e., it will remain an 8-bit value).

String literals for the various types of unicode encodings (and associated variables) can be defined as follows:

```
char     utf_8[] = u8"This is UTF-8 encoded.";
char16_t utf16[] = u"This is UTF-16 encoded.";
char32_t utf32[] = U"This is UTF-32 encoded.";
```

Alternatively, unicode constants may be defined using the `\u` escape sequence, followed by a hexadecimal value. Depending on the type of the unicode variable (or constant) a `UTF-8, UTF-16` or `UTF-32` value is used. E.g.,

```
char     utf_8[] = u8"\u2018";
char16_t utf16[] = u"\u2018";
char32_t utf32[] = U"\u2018";
```

Unicode strings can be delimited by double quotes but raw string literals can also be used.

3.4.4 The data type 'long long int' (C++0x)

The C++0x standard adds the type `long long int` to the set of standard types. On 32 bit systems it will have at least 64 usable bits. Some compilers already supported `long long int` as an extension, but C++0x officially adds it to **C++**.

3.4.5 The data type 'size_t'

The `size_t` type is not really a built-in primitive data type, but a data type that is promoted by **POSIX** as a typename to be used for non-negative integral values answering questions like 'how much' and 'how

many', in which case it should be used instead of `unsigned int`. It is not a specific **C++** type, but also available in, e.g., **C**. Usually it is defined implictly when a (any) system header file is included. The header file 'officially' defining `size_t` in the context of **C++** is `cstddef`.

Using `size_t` has the advantage of being a *conceptual* type, rather than a standard type that is then modified by a modifier. Thus, it improves the self-documenting value of source code.

Sometimes functions explictly require `unsigned int` to be used. E.g., on amd-architectures the X-windows function `XQueryPointer` explicitly requires a pointer to an `unsigned int` variable as one of its arguments. In such situations a pointer to a `size_t` variable can't be used, but the address of an `unsigned int` must be provided. Such situations are exceptional, though.

Other useful bit-represented types also exists. E.g., `uint32_t` is guaranteed to hold 32-bits unsigned values. Analogously, `int32_t` holds 32-bits signed values. Corresponding types exist for 8, 16 and 64 bits values. These types are defined in the header file `cstdint`.

3.5 A new syntax for casts

Traditionally, **C** offers the following *cast* syntax:

```
(typename)expression
```

here `typename` is the name of a valid *type*, and `expression` is an expression.

C style casts are now deprecated. Although **C++** offers *function call* notations using the following syntax:

```
typename(expression)
```

this syntax not actually represents a cast, but a request to the compiler to construct an (anonymous) variable having type `typename` from `expression`. Although this form is very often used in **C++**, it should *not* be used for casting. Instead, there are now four *new-style casts* available, that are introduced in the following sections.

3.5.1 The 'static_cast'-operator

The cast converting conceptually comparable types to each other is:

```
static_cast<type>(expression)
```

This type of cast is used to convert, e.g., a `double` to an `int`: both are numbers, but as the `int` has no fractions precision is potentially reduced. But the converse also holds true. When the quotient of two `int` values must be assigned to a `double` the fraction part of the division will get lost unless a cast is used.

Here is an example of such a cast is (assuming `quotient` is of type `double` and `lhs` and `rhs` are `int`-typed variables):

```
quotient = static_cast<double>(lhs) / rhs;
```

If the cast is omitted, the division operator will ignore the remainder as its operands are `int` expressions. Note that the division should be put outside of the cast expression. If the division is put inside (as in `static_cast<double>(lhs / rhs)`) an *integer division* will have been performed *before* the cast has had a chance to convert the type of an operand to `double`.

Another nice example of code in which it is a good idea to use the `static_cast<>()`-operator is in situations where the arithmetic assignment operators are used in mixed-typed exprressions. Consider the following expression (assume `doubleVar` is a variable of type `double`):

```
intVar += doubleVar;
```

This statement actually evaluates to:

```
intVar = static_cast<int>(static_cast<double>(intVar) +
doubleVar);
```

Here `IntVar` is first promoted to a `double`, and is then added as a `double` value to `doubleVar`. Next, the sum is cast back to an `int`. These two casts are a bit overdone. The same result is obtained by explicitly casting `doubleVar` to an `int`, thus obtaining an `int`-value for the right-hand side of the expression:

```
intVar += static_cast<int>(doubleVar);
```

A `static_cast` can also be used to undo or introduce the signed-modifier of an `int`-typed variable. The **C** function `tolower` requires an `int` representing the value of an `unsigned char`. But `char` by default is a signed type. To call `tolower` using an available `char ch` we should use:

```
tolower(static_cast<unsigned char>(ch))
```

Casts like these provide information to the compiler about how to handle the provided data. Very often (especially with data types differing only in size but not in representation) the cast won't require any additional code. Additional code will be required, however, to convert one representation to another, e.g., when converting `double` to `int`.

3.5.2 The 'const_cast'-operator

The `const` keyword has been given a special place in casting. Normally anything `const` is `const` for a good reason. Nonetheless situations may be encountered where the `const` can be ignored. For these special situations the `const_cast` should be used. Its syntax is:

```
const_cast<type>(expression)
```

A `const_cast<type>(expression)` expression is used to undo the `const` attribute of a (pointer) type.

The need for a `const_cast` may occur in combination with functions from the standard **C** library which traditionally weren't always as const-aware as they should. A function `strfun(char *s)` might be available, performing some operation on its `char *s` parameter without actually modifying the characters pointed to by `s`. Passing `char const hello[] = "hello";` to `strfun` will produce the warning

```
passing 'const char *' as argument 1 of 'fun(char *)' discards const
```

A `const_cast` is the appropriate way to prevent the warning:

```
strfun(const_cast<char *>(hello));
```

3.5.3 The 'reinterpret_cast'-operator

The third new-style cast is used to change the *interpretation* of information: the `reinterpret_cast`. It is somewhat reminiscent of the `static_cast`, but `reinterpret_cast` should be used when it is *known* that the information as defined in fact is or can be interpreted as something completely different. Its syntax is:

```
reinterpret_cast<pointer type>(pointer expression)
```

A `reinterpret_cast<type>(expression)` operator is appropriately used to reinterpret a `void *` to a pointer of a well-known type. Void pointers are encountered with functions from the **C** library like `qsort`. The `qsort` function expects a pointer to a (comparison) function having two `void const *` parameters. In fact, the `void const *`s point to data elements of the array to sort, and so the comparison function may cast the `void const *` parameters to pointers to the elements of the array to be sorted. E.g., if the array is an `int array[]` and the compare function's parameters are `void const *p1, void const *p2` then the compare function may obtain the address of the `int` pointed to by `p1` by using:

```
reinterpret_cast<int const *>(p1)
```

Another example of a `reinterpret_cast` is found in combination with the `write` functions that are available for files and streams. In **C++** streams are the preferred interface to, e.g., files. Output streams (like `cout`) offer `write` members having the prototype

```
write(char const *buffer, int length)
```

To write a `double` to a stream using `write` a `reinterpret_cast` is needed as well. E.g., to write the raw bytes of a variable `double value` to `cout` we would use:

```
cout.write(reinterpret_cast<char const *>(&value), sizeof(double));
```

All casts are potentially dangerous, but the `reinterpret_cast` is the most dangerous of all casts. Effectively we tell the compiler: back off, we know what we're doing, so stop fuzzing. All bets are off, and we'd better *do* know what we're doing in situations like these. As a case in point consider the following code:

```
int value = 0x12345678;     // assume a 32-bits int

cout << "Value's first byte has value: " << hex <<
        static_cast<int>(
            *reinterpret_cast<unsigned char *>(&value)
                        );
```

The above code will show different results on little and big endian computers. Little endian computers will show the value 78, big endian computers the value 12. Also note that the different representations used by little and big endian computers renders the previous example (`cout.write(...)`) non-portable over computers of different architectures.

As a rule of thumb: if circumstances arise in which casts *have* to be used, clearly document the reasons for their use in your code, making double sure that the cast will not eventually cause a program to misbehave.

3.5.4 The 'dynamic_cast'-operator

Finally there is a new style cast that is used in combination with polymorphism (see chapter 14). Its syntax is:

```
dynamic_cast<type>(expression)
```

It is used run-time to convert, a pointer to an object of a class to a pointer to an object of a class that is found further down its so-called *class hierarchy* (which is also called a *downcast*). At this point in the *Annotations* a `dynamic_cast` cannot yet be discussed extensively, but we will return to this topic in section 14.5.1.

3.6 Keywords and reserved names in C++

C++'s keywords are a superset of **C**'s keywords. Here is a list of all keywords of the language:

```
alignof compl        explicit  new              short       typename
and     concept      extern    not              signed      union
and_eq  const        false     not_eq           sizeof      unsigned
asm     const_cast   float     nullptr          static      using
auto    constexpr    for       operator         static_cast virtual
axiom   continue     friend    or               struct      void
bitand  decltype     goto      or_eq            switch      volatile
bitor   default      if        private          template    wchar_t
bool    delete       import    protected        this        while
break   do           inline    public           throw       xor
case    double       int       register         true        xor_eq
catch   dynamic_cast long      reinterpret_cast try
char    else         mutable   requires         typedef
class   enum         namespace return           typeid
```

Notes:

- The `nullptr` keyword is defined in the C++0x standard (not yet supported by the `g++` compiler).
- the *operator keywords*: `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor` and `xor_eq` are symbolic alternatives for, respectively, `&&`, `&=`, `&`, `|`, `~`, `!`, `!=`, `||`, `|=`, `^` and `^=`.

Keywords can only be used for their intended purpose and cannot be used as names for other entities (e.g., variables, functions, class-names, etc.). In addition to keywords identifiers starting with an underscore and living in the *global namespace* (i.e., not using any explicit namespace or using the mere `::` namespace specification) or living in the *std namespace* are reserved identifiers in the sense that their use is a prerogative of the implementor.

Chapter 4

Name Spaces

4.1 Namespaces

Imagine a math teacher who wants to develop an interactive math program. For this program functions like `cos, sin, tan` etc. are to be used accepting arguments in degrees rather than arguments in radians. Unfortunately, the function name `cos` is already in use, and that function accepts radians as its arguments, rather than degrees.

Problems like these are usually solved by defining another name, e.g., the function name `cosDegrees` is defined. **C++** offers an alternative solution through *namespaces*. Namespaces can be considered as areas or regions in the code in which identifiers are defined that normally won't conflict with names already defined elsewhere.

Now that the ANSI/ISO standard has been implemented to a large degree in recent compilers, the use of namespaces is more strictly enforced than in previous versions of compilers. This affects the setup of `class` header files. At this point in the Annotations this cannot be discussed in detail, but in section 7.9.1 the construction of header files using entities from namespaces is discussed.

4.1.1 Defining namespaces

Namespaces are defined according to the following syntax:

```
namespace identifier
{
    // declared or defined entities
    // (declarative region)
}
```

The identifier used when defining a namespace is a standard **C++** identifier.

Within the *declarative region*, introduced in the above code example, functions, variables, structs, classes and even (nested) namespaces can be defined or declared. Namespaces cannot be defined within a function body. However, it is possible to define a namespace using multiple *namespace* declarations. Namespaces are '*open*' meaning that a namespace `CppAnnotations` could be defined in a file `file1.cc` and also in a file `file2.cc`. Entities defined in the `CppAnnotations` namespace of files `file1.cc` and `file2.cc` are then united in one `CppAnnotations` namespace region. For example:

```
// in file1.cc
namespace CppAnnotations
{
    double cos(double argInDegrees)
```

```
        {
            ...
        }
    }

    // in file2.cc
    namespace CppAnnotations
    {
        double sin(double argInDegrees)
        {
            ...
        }
    }
```

Both `sin` and `cos` are now defined in the same `CppAnnotations` namespace.

Namespace entities can be defined outside of their namespaces. This topic is discussed in section 4.1.4.1.

4.1.1.1 Declaring entities in namespaces

Instead of *defining* entities in a namespace, entities may also be *declared* in a namespace. This allows us to put all the declarations in a header file that can thereupon be included in sources using the entities defined in the namespace. Such a header file could contain, e.g.,

```
namespace CppAnnotations
{
    double cos(double degrees);
    double sin(double degrees);
}
```

4.1.1.2 A closed namespace

Namespaces can be defined without a name. Such an anonymous namespace restricts the visibility of the defined entities to the source file defining the anonymous namespace.

Entities defined in the anonymous namespace are comparable to **C**'s `static` functions and variables. In **C++** the `static` keyword can still be used, but its preferred use is in `class` definitions (see chapter 7). In situations where in **C** static variables or functions would have been used the anonymous namespace should be used in **C++**.

The anonymous namespace is a closed namespace: it is not possible to add entities to the same anonymous namespace using different source files.

4.1.2 Referring to entities

Given a namespace and its entities, the scope resolution operator can be used to refer to its entities. For example, the function `cos()` defined in the `CppAnnotations` namespace may be used as follows:

```
// assume CppAnnotations namespace is declared in the
// following header file:
#include <cppannotations>

int main()
{
```

```
        cout << "The cosine of 60 degrees is: " <<
                CppAnnotations::cos(60) << '\n';
    }
```

This is a rather cumbersome way to refer to the `cos()` function in the `CppAnnotations` namespace, especially so if the function is frequently used. In cases like these an *abbreviated* form can be used after specifying a *using declaration*. Following

```
    using CppAnnotations::cos;  // note: no function prototype,
                                // just the name of the entity
                                // is required.
```

calling `cos` will call the `cos` function defined in the `CppAnnotations` namespace. This implies that the standard `cos` function, accepting radians, is not automatically called anymore. To call that latter `cos` function the plain scope resolution operator should be used:

```
    int main()
    {
        using CppAnnotations::cos;
        ...
        cout << cos(60)         // calls CppAnnotations::cos()
            << ::cos(1.5)       // call the standard cos() function
            << '\n';
    }
```

A `using` declaration can have restricted scope. It can be used inside a block. The `using` declaration prevents the definition of entities having the same name as the one used in the `using` declaration. It is not possible to specify a `using` declaration for a variable `value` in some namespace, and to define (or declare) an identically named object in a block also containing a `using` declaration. Example:

```
    int main()
    {
        using CppAnnotations::value;
        ...
        cout << value << '\n';  // uses CppAnnotations::value
        int value;              // error: value already declared.
    }
```

4.1.2.1 The 'using' directive

A generalized alternative to the `using` declaration is the *using directive*:

```
    using namespace CppAnnotations;
```

Following this directive, *all* entities defined in the `CppAnnotations` namespace are used as if they were declared by `using` declarations.

While the `using` directive is a quick way to import all the names of a namespace (assuming the namespace has previously been declared or defined), it is at the same time a somewhat dirty way to do so, as it is less clear what entity will be used in a particular block of code.

If, e.g., `cos` is defined in the `CppAnnotations` namespace, `CppAnnotations::cos` will be used when `cos` is called. However, if `cos` is *not* defined in the `CppAnnotations` namespace, the standard `cos` function will be used. The `using` directive does not document as clearly as the `using` declaration what entity will be used. Therefore use caution when applying the `using` directive.

4.1.2.2 'Koenig lookup'

If *Koenig lookup* were called the 'Koenig principle', it could have been the title of a new Ludlum novel. However, it is not. Instead it refers to a **C++** technicality.

'Koenig lookup' refers to the fact that if a function is called without secifying its namespace, then the namespaces of its arguments are used to determine its namespace. If the namespace in which the arguments are defined contains such a function, then that function is used. This is called the 'Koenig lookup'.

This is illustrated in the following example. The function `FBB::fun(FBB::Value v)` is defined in the `FBB` namespace. It can be called without explicitly mentioning its namespace:

```
#include <iostream>

namespace FBB
{
    enum Value        // defines FBB::Value
    {
        FIRST
    };

    void fun(Value x)
    {
        std::cout << "fun called for " << x << '\n';
    }
}

int main()
{
    fun(FBB::FIRST);    // Koenig lookup: no namespace
                        // for fun() specified
}
/*
    generated output:
fun called for 0
*/
```

The compiler is fairly smart when handling namespaces. If `Value` in the `namespace FBB` would have been defined as `typedef int Value` then `FBB::Value` would be recognized as `int`, thus causing the Koenig lookup to fail.

As another example, consider the next program. Here two namespaces are involved, each defining their own `fun` function. There is no ambiguity, since the argument defines the namespace and `FBB::fun` is called:

```
#include <iostream>

namespace FBB
{
    enum Value        // defines FBB::Value
    {
        FIRST
    };

    void fun(Value x)
    {
        std::cout << "FBB::fun() called for " << x << '\n';
    }
```

```
}

namespace ES
{
    void fun(FBB::Value x)
    {
        std::cout << "ES::fun() called for " << x << '\n';
    }
}

int main()
{
    fun(FBB::FIRST);    // No ambiguity: argument determines
                        // the namespace
}
/*
    generated output:
FBB::fun() called for 0
*/
```

Here is an example in which there *is* an ambiguity: `fun` has two arguments, one from each namespace. The ambiguity must be resolved by the programmer:

```
#include <iostream>

namespace ES
{
    enum Value        // defines ES::Value
    {
        FIRST
    };
}

namespace FBB
{
    enum Value        // defines FBB::Value
    {
        FIRST
    };

    void fun(Value x, ES::Value y)
    {
        std::cout << "FBB::fun() called\n";
    }
}

namespace ES
{
    void fun(FBB::Value x, Value y)
    {
        std::cout << "ES::fun() called\n";
    }
}

int main()
{
    //  fun(FBB::FIRST, ES::FIRST); ambiguity: resolved by
    //                              explicitly mentioning
    //                              the namespace
```

```
    ES::fun(FBB::FIRST, ES::FIRST);
}
/*
    generated output:
ES::fun() called
*/
```

An interesting subtlety with namespaces is that definitions in one namespace may break the code defined in another namespace. It shows that namespaces may affect each other and that may backfire if we're not aware of their peculiarities. Consider the following example:

```
namespace FBB
{
    struct Value
    {};

    void fun(int x);
    void gun(Value x);
}

namespace ES
{
    void fun(int x)
    {
        fun(x);
    }
    void gun(FBB::Value x)
    {
        gun(x);
    }
}
```

Whatever happens, the programmer'd better not use any of the `ES::fun` functions since it will doubtlessly result in infinite recursion, but that's not the point. The point is that the programmer won't even be given the opportunity to call `ES::fun` since the compilation fails.

Compilation fails for `gun` but not for `fun`. Why is that? Why is `ES::fun` flawlessly compiling while `ES::gun` isn't? In `ES::fun` `fun(x)` is called. As `x`'s type is not defined in a namespace the Koenig lookup does not apply and `fun` calls itself with infinite recursion.

With `ES::gun` the argument is defined in the `FBB` namespace. Consequently, the `FBB::gun` function is a possible candidate to be called. But `ES::gun` itself also is possible as `ES::gun`'s prototype perfectly matches the call `gun(x)`.

Now consider the situation where `FBB::gun` has not yet been declared. Then there is of course no ambiguity. The programmer responsible for the `ES` namespace is resting happily. Some time after that the programmer who's maintaining the `FBB` namespace decides it may be nice to add a function `gun(Value x)` to the `FBB` namespace. Now suddenly the code in the namespace `ES` breaks because of an addition in a completely other namespace (`FBB`). Namespaces clearly are not completely independent of each other and we should be aware of subtleties like the above. Later in the **C++** Annotations (chapter 10) we'll return to this issue.

4.1.3 The standard namespace

The `std` namespace is reserved by **C++**. The standard defines many entities that are part of the runtime available software (e.g., `cout, cin, cerr`); the templates defined in the *Standard Template Library* (cf. chapter 18); and the *Generic Algorithms* (cf. chapter 19) are defined in the `std` namespace.

Regarding the discussion in the previous section, `using` declarations may be used when referring to entities in the `std` namespace. For example, to use the `std::cout` stream, the code may declare this object as follows:

```
#include <iostream>
using std::cout;
```

Often, however, the identifiers defined in the `std` namespace can all be accepted without much thought. Because of that, one frequently encounters a `using` directive, allowing the programmer to omit a namespace prefix when referring to any of the entities defined in the namespace specified with the `using` directive. Instead of specifying `using` declarations the following `using` directive is frequently encountered: construction like

```
#include <iostream>
using namespace std;
```

Should a `using` directive, rather than `using` declarations be used? As a rule of thumb one might decide to stick to `using` declarations, up to the point where the list becomes impractically long, at which point a `using` directive could be considered.

Two restrictions apply to `using` directives and declarations:

- Programmers should not declare or define anything inside the `namespace std`. This is *not* compiler enforced but is imposed upon user code by the standard;
- `Using` declarations and directives should not be imposed upon code written by third parties. In practice this means that `using` directives and declarations should be banned from header files and should only be used in source files (cf. section 7.9.1).

4.1.4 Nesting namespaces and namespace aliasing

Namespaces can be nested. Here is an example:

```
namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;
    }
}
```

The variable `pointer` is defined in the `Virtual` namespace, that itself is nested under the `CppAnnotations` namespace. To refer to this variable the following options are available:

- The *fully qualified name* can be used. A fully qualified name of an entity is a list of all the namespaces that are encountered until reaching the definition of the entity. The namespaces and entity are glued together by the scope resolution operator:

  ```
  int main()
  {
      CppAnnotations::Virtual::pointer = 0;
  }
  ```

- A `using` declaration for `CppAnnotations::Virtual` can be provided. Now `Virtual` can be used without any prefix, but `pointer` must be used with the `Virtual::` prefix:

  ```
  using CppAnnotations::Virtual;
  ```

```
int main()
{
    Virtual::pointer = 0;
}
```

- A `using` declaration for `CppAnnotations::Virtual::pointer` can be used. Now `pointer` can be used without any prefix:

```
using CppAnnotations::Virtual::pointer;

int main()
{
    pointer = 0;
}
```

- A `using` directive or directives can be used:

```
using namespace CppAnnotations::Virtual;

int main()
{
    pointer = 0;
}
```

  Alternatively, two separate `using` directives could have been used:

```
using namespace CppAnnotations;
using namespace Virtual;

int main()
{
    pointer = 0;
}
```

- A combination of `using` declarations and `using` directives can be used. E.g., a `using` directive can be used for the `CppAnnotations` namespace, and a `using` declaration can be used for the `Virtual::pointer` variable:

```
using namespace CppAnnotations;
using Virtual::pointer;

int main()
{
    pointer = 0;
}
```

At every `using` directive all entities of that namespace can be used without any further prefix. If a namespace is nested, then that namespace can also be used without any further prefix. However, the entities defined in the nested namespace still need the nested namespace's name. Only after applying a `using` declaration or directive the qualified name of the nested namespace can be omitted.

When fully qualified names are preferred but a long name like

```
CppAnnotations::Virtual::pointer
```

is nevertheless considered too long, a *namespace alias* may be used:

```
namespace CV = CppAnnotations::Virtual;
```

This defines CV as an *alias* for the full name. The variable pointer may now be accessed using:

```
CV::pointer = 0;
```

A namespace alias can also be used in a using declaration or directive:

```
namespace CV = CppAnnotations::Virtual;
using namespace CV;
```

4.1.4.1 Defining entities outside of their namespaces

It is not strictly necessary to define members of namespaces inside a namespace region. But before an entity is defined *outside* of a namespace it must have been declared *inside* its namespace.

To define an entity outside of its namespace its name must be *fully qualified* by prefixing the member by its namespaces. The definition may be provided at the global level or at intermediate levels in the case of nested namespaces. This allows us to define an entity belonging to namespace A::B within the region of namespace A.

Assume the type int INT8[8] is defined in the CppAnnotations::Virtual namespace. Furthermore assume that is is our intent to define a function squares, inside the namespace CppAnnotations::Virtual returning a pointer to CppAnnotations::Virtual::INT8.

Having defined the prerequisites within the CppAnnotations::Virtual namespace, our function could be defined as follows (cf. chapter 8 for coverage of the memory allocation operator new[]):

```
namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;

        typedef int INT8[8];

        INT8 *squares()
        {
            INT8 *ip = new INT8[1];

            for (size_t idx = 0; idx != sizeof(INT8) / sizeof(int); ++idx)
                (*ip)[idx] = (idx + 1) * (idx + 1);

            return ip;
        }
    }
}
```

The function squares defines an array of one INT8 vector, and returns its address after initializing the vector by the squares of the first eight natural numbers.

Now the function squares can be defined outside of the CppAnnotations::Virtual namespace:

```
namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;
```

```
            typedef int INT8[8];

            INT8 *squares();
        }
    }

    CppAnnotations::Virtual::INT8 *CppAnnotations::Virtual::squares()
    {
        INT8 *ip = new INT8[1];

        for (size_t idx = 0; idx != sizeof(INT8) / sizeof(int); ++idx)
            (*ip)[idx] = (idx + 1) * (idx + 1);

        return ip;
    }
```

In the above code fragment note the following:

- `squares` is declared inside of the `CppAnnotations::Virtual` namespace.
- The definition outside of the namespace region requires us to use the fully qualified name of the function *and* of its return type.
- *Inside* the body of the function `squares` we are within the `CppAnnotations::Virtual` namespace, so inside the function fully qualified names (e.g., for `INT8`) are not required any more.

Finally, note that the function could also have been defined in the `CppAnnotations` region. In that case the `Virtual` namespace would have been required when defining `squares()` and when specifying its return type, while the internals of the function would remain the same:

```
    namespace CppAnnotations
    {
        namespace Virtual
        {
            void *pointer;

            typedef int INT8[8];

            INT8 *squares();
        }

        Virtual::INT8 *Virtual::squares()
        {
            INT8 *ip = new INT8[1];

            for (size_t idx = 0; idx != sizeof(INT8) / sizeof(int); ++idx)
                (*ip)[idx] = (idx + 1) * (idx + 1);

            return ip;
        }
    }
```

Chapter 5

The 'string' Data Type

C++ offers many solutions for common problems. Most of these facilities are part of the *Standard Template Library* or they are implemented as *generic algorithms* (see chapter 19).

Among the facilities **C++** programmers have developed over and over again are those manipulating chunks of text, commonly called *strings*. The **C** programming language offers rudimentary string support. **C**'s *ASCII-Z* terminated series of characters form the foundation upon which an enourmous amount of code has been built[1].

To process text **C++** offers a `std::string` type. In **C++** the traditional **C** library functions manipulating ASCII-Z strings are deprecated in favor of using `string` objects. Many problems in **C** programs are caused by buffer overruns, boundary errors and allocation problems that can be traced back to improperly using these traditional **C** string library functions. Many of these problems can be prevented using **C++** string objects.

Actually, `string` objects are *class type* variables, and in that sense they are comparable to stream objects like `cin` and `cout`. In this section the use of `string` type objects is covered. The focus is on their definition and their use. When using `string` objects the *member function syntax* is commonly used:

```
stringVariable.operation(argumentList)
```

For example, if `string1` and `string2` are variables of type `std::string`, then

```
string1.compare(string2)
```

can be used to compare both strings.

In addition to the common member functions the `string` class also offers a wide variety of *operators*, like the assignment (`=`) and the comparison operator (`==`). Operators often result in code that is easy to understand and their use is generally preferred over the use of member functions offering comparable functionality. E.g., rather than writing

```
if (string1.compare(string2) == 0)
```

the following is generally preferred:

```
if (string1 == string2)
```

To define and use `string`-type objects, sources must include the header file `string`. To merely *declare* the string type the header should be included.

[1] We define an ASCII-Z string as a series of ASCII-characters terminated by the ASCII-character zero (hence -Z), which has the value zero, and should not be confused with character `'0'`, which usually has the value `0x30`

5.1 Operations on strings

Some of the operations that can be performed on strings return indices within the strings. Whenever such an operation fails to find an appropriate index, the *value* `string::npos` is returned. This value is a symbolic value of type `string::size_type`, which is (for all practical purposes) an (`unsigned`) `int`.

All `string` members accepting `string` objects as arguments also accept `char const *` (**C** ASCII-Z string) arguments. The same usually holds true for operators accepting `string` objects.

Some `string`-members use *iterators*. Iterators are formally introduced in section 18.2. Member functions using iterators are listed in the next section (5.2), but the iterator concept itself is not further covered by this chapter.

Strings support a large variety of members and operators. A short overview listing their capabilities is provided in this section, with subsequent sections offering a detailed discussion. The bottom line: **C++** strings are extremely versatile and there is hardly a reason for falling back on the **C** library to process text. **C++** strings handle all the required memory management and thus memory related problems, which is the #1 source of problems in **C** programs, can be prevented when **C++** strings are used. Strings do come at a price, though. The class's extensive capabilities have also turned it into a beast. It's hard to learn and master all its features and in the end you'll find that not all that you expected is actuallly there. For example, `std::string` doesn't offer case-insensitive comparisons. But in the end it even isn't as simple as that. It *is* there, but it is somewhat hidden and at this point in the **C++** Annotations it's too early to study into that hidden corner yet. Instead, realize that **C**'s standard library *does* offer useful functions that can be used as long as we're aware of their limitations and are able to avoid their traps. So for now, to perform a classical case-insensitive comparison of the contents of two `std::string` objects `str1` and `str2` the following will do:

```
strcasecmp(str1.c_str(), str2.c_str());
```

Strings support the following functionality:

- `initialization`:

  when string objects are defined they are always properly initialized. In other words, they are always in a valid state. Strings may be initialized empty or already existing text can be used to initialize strings.

- `assignment`:

  strings may be given new values. New values may be assigned using member functions (like `assign`) but a plain assignment operator (i.e., =)may also be used. Furthermore, assignment *to* a character buffer is also supported.

- `conversions`:

  the partial or complete contents of string objects may be interpreted as **C** strings but the string's contents may also be processed as a series of raw binary bytes, not necessarily terminating in an ASCII-Z byte. Furthermore, in many situations plain characters and **C** strings may be used where `std::string`s are accepted as well.

- `breakdown`:

  the individual characters stored in a string can be accessed using the familiar index operator (`[]`) allowing us to either access or modify information in the middle of a string.

- `comparisons`:

  strings may be compared to other strings (or **C** ASCII-Z strings) using the familiar logical comparison operators `==`, `!=`, `<`, `<=`, `>` and >=. There are also member functions available offering a more fine-grained comparison.

- `modification`:

  the contents of strings may me modified in many ways. Operators are available to add information to string objects, to insert information in the middle of string objects, or to replace or erase (parts of) a string's contents.

- `swapping`:

  the string's swapping capability allows us in principle to exchange the contents of two string objects without a byte-by-byte copying operation of the string's contents.

- `searching`:

  the locations of characters, sets of characters, or series of characters may be searched for from any position within the string object and either searching in a forward or backward direction.

- `housekeeping`:

  several housekeeping facilities are offered: the string's length, or its empty-state may be interrogated. But string objects may also be resized.

- `stream I/O`:

  strings may be extracted from or inserted into streams. In addition to plain string extraction a line of a text file may be read without running the risk of a buffer overrun. Since extraction and insertion operations are stream based the I/O facilities are *device independent*.

5.2 A std::string reference

In this section the string members and string-related operations are referenced. The subsections cover, respectively the string's initializers, iterators, operators, and member functions. The following terminology is used throughout this section:

- `object` is always a `string`-object;
- `argument` is a `string const &` or a `char const *` unless indicated otherwise. The contents of an `argument` never is modified by the operation processing the `argument`;
- `opos` refers to an offset into an `object` string;
- `apos` refers to an offset into an `argument`;
- `on` represents a number of characters in an `object` (starting at `opos`);
- `an` represents a number of characters in an `argument` (starting at `apos`).

Both `opos` and `apos` must refer to existing offsets, or an exception (cf. chapter 9) is generated. In contrast, `an` and `on` may exceed the number of available characters, in which case only the available characters will be considered.

Many members declare default values for `on, an` and `apos`. Some members declare default values for `opos`. Default offset values are 0, the default values of `on` and `an` is `string::npos`, which can be interpreted as 'the required number of characters to reach the end of the string'.

With members starting their operations at the end of the string object's contents proceeding backwards, the default value of `opos` is the index of the object's *last* character, with `on` by default equal to `opos + 1`, representing the length of the substring *ending* at `opos`.

In the overview of member functions presented below it may be assumed that all these parameters accept default values unless indicated otherwise. Of course, the default argument values cannot be used if a function requires additional arguments beyond the ones otherwise accepting default values.

Some members have overloaded versions expecting an initial argument of type `char const *`. But even if that is not the case the first argument can *always* be of type `char const *` where a parameter of `std::string` is defined.

Several member functions accept *iterators*. Section 18.2 covers the technical aspects of *iterators*, but these may be ignored at this point without loss of continuity. Like `apos` and `opos`, iterators must refer to existing positions and/or to an existing range of characters within the string object's contents.

Finally, all `string`-member functions computing indices return the predefined constant `string::npos` on failure.

5.2.1 Initializers

After defining string objects they are guaranteed to be in a valid state. At *definition time* string objects may be initialized in one of the following ways: The following `string` constructors are available:

- `string object`:

  initializes `object` to an empty string. When defining a `string` this way no argument list may be specified;

- `string object(string::size_type count, char ch)`:

  initializes `object` with `count` characters `ch`;

- `string object(string const &argument)`:

  initializes `object` with `argument`;

- `string object(std::string const &argument, string::size_type apos, string::size_type an)`:

  initializes `object` with `argument`'s contents starting at index position `apos`, using at most `an` of `argument`'s characters;

- `string object(InputIterator begin, InputIterator end)`:

  initializes `object` with the characters in the range of characters defined by the two `InputIterators`.

5.2.2 Iterators

See section 18.2 for details about *iterators*. As a quick introduction to iterators: an iterator acts like a pointer, and pointers can often be used in situations where iterators are requested. Iterators usually come in pairs, defining a range of entities. The begin-iterator points to the first entity of the range, the end-iterator points just beyond the last entity that will be considered. Their difference is equal to the number of entities in the iterator-range.

Iterators play an important role in the context of *generic algorithms* (cf. chapter 19). The class `std::string` defines the following *iterator types*:

- `string::iterator` and `string::const_iterator`:

  these iterators are *forward iterators*. The `const_iterator` is returned by `string const` objects, the plain `iterator` is returned by non-const string objects. Characters referred to by `iterators` may be modified;

- `string::reverse_iterator` and `string::reverse_const_iterator`:

  these iterators are also *forward iterators* but when *incrementing* the iterator the *previous* character in the string object is reached. Other than that they are comparable to, respectively, `string::iterator` and `string::const_iterator`.

5.2.3 Operators

String objects may be manipulated by member functions but also by operators. Using operators often results in more natural-looking code. In cases where operators are available having equivalent functionality as member function the operator is practically always preferred.

The following operators are available for `string` objects (in the examples 'object' and 'argument' refer to existing `std::string` objects).

- plain assignment:

  a character, **C** or **C++** string may be assigned to a `string` object. The assignment operator returns its left-hand side operand. Example:

  ```
  object =  argument;
  object = "C string";
  object = 'x';
  object = 120;       // same as object = 'x'
  ```

- addition:

  the arithmetic additive assignment operator and the addition operator add text to a `string` object. The arithmetic assignment operator returns its left-hand side operand, the addition operator returns its result in a temporary string object. When using the addition operator either the left-hand side operand or the right-hand side operand must be a `std::string` object. The other operand may be a char, a **C** string or a **C++** string. Example:

  ```
  object += argument;
  object += "hello";
  object += 'x';      // integral expressions are OK

  argument + otherArgument;   // two std::string objects
  argument + "hello";         // using + at least one
  "hello" + argument;         // std::string is required
  argument + 'a'              // integral expressions are OK
  'a' + argument;
  ```

- index operator:

  The index operator may be used to retrieve `object`'s individual characters, or to assign new values to individual characters of a non-const string object. There is no range-checking (use the `at()` member function for that). This operator returns a `char &` or `char const &`. Example:

  ```
  object[3] = argument[5];
  ```

- logical operators:

  the logical comparison operators may be applied to two string objects or to a string object and a **C** string to compare their contents. These operators return a `bool` value The `==`, `!=`, `>`, `>=`, `<` and `<=` operators are available. The ordering operators perform a lexicographical comparison of their contents using the ASCII character collating sequence. Example:

  ```
  object == object;           // true
  object != (object + 'x');   // true
  object <= (object + 'x');   // true
  ```

- stream related operators:

  the insertion-operator (cf. section 3.1.4) may be used to insert a `string` object into an `ostream`, the extraction-operator may be used to extract a string object from an `istream`. The extraction operator by default first ignores all white space characters

and then extracts all consecutively non-blank characters from an `istream`. Instead of a string a character array may be extracted as well, but the advantage of using a string object should be clear: the destination string object is automatically resized to the required number of characters. Example:

```
cin >> object;
cout << object;
```

5.2.4 Member functions

The `std::string` class offers many member function as well as additional non-member functions that should be considered part of the string class. All these functions are listed below in alphabetic order.

The symbolic value `string::npos` is defined by the string class. It represents 'index-not-found' when returned by member functions returning string offset positions. Example: when calling '`object.find('x')`' (see below) on a string object not containing the character `'x'`, `npos` is returned, as the requested position does not exist.

The final 0-byte used in **C** strings to indicate the end of an ASCII-Z string is *not* considered part of a **C++** string, and so the member function will return `npos`, rather than `length()` when looking for 0 in a string object containing the characters of a **C** string.

Here are the standard functions that operate on objects of the class string. When a parameter of `size_t` is mentioned it may be interpreted as a parameter of type `string::size_type`, but without defining a default argument value. The type `size_type` should be read as `string::size_type`. With `size_type` the default argument values mentioned in section 5.2 apply. All quoted functions are member functions of the class `std::string`, except where indicated otherwise.

- `char &at(size_t opos)`:

  a reference to the character at the indicated position is returned. When called with `string const` objects a `char const &` is returned. The member function performs range-checking, raising an exception (that by default aborts the program) if an invalid index is passed.

- `string &append(InputIterator begin, InputIterator end)`:

  the characters in the range defined by `begin` and `end` are appended to the current string object.

- `string &append(string const &argument, size_type apos, size_type an)`:

  `argument` (or a substring) is appended to the current string object.

- `string &append(char const *argument, size_type an)`:

  the first `an` characters of `argument` are appended to the string object.

- `string &append(size_type n, char ch)`:

  `n` characters `ch` are appended to the current string object.

- `string &assign(string const &argument, size_type apos, size_type an)`:

  `argument` (or a substring) is assigned to the string object. If `argument` is of type `char const *` and one additional argument is provided the second argument is interpreted as a value initializing `an`, using 0 to initialize `apos`.

- `string &assign(size_type n, char ch)`:

  `n` characters `ch` are assigned to the current string object.

- `size_type capacity() const`:

  the number of characters that can currently be stored in the string object without needing to resize it is returned.

- `int compare(string const &argument) const`:

  the text stored in the current string object and the text stored in `argument` is compared using a lexicographical comparison using the ASCII character collating sequence. zero is returned if the two strings have identical contents, a negative value is returned if the text in the current object should be ordered *before* the text in `argument`; a positive value is returned if the text in the current object should be ordered *beyond* the text in `argument`.

- `int compare(size_t opos, size_t on, string const &argument) const`:

  a substring of the text stored in the current string object is compared to the text stored in `argument`. At most `on` characters starting at offset `opos` are compared to the text in `argument`.

- `int compare(size_t opos, size_t on, string const &argument, size_type apos, size_type an)`:

  a substring of the text stored in the current string object is compared to a substring of the text stored in `argument`. At most `on` characters of the current string object, starting at offset `opos`, are compared to at most `an` characters of `argument`, starting at offset `apos`. In this case `argument` *must* be a string object.

- `int compare(size_t opos, size_t on, char const *argument, size_t an)`:

  a substring of the text stored in the current string object is compared to a substring of the text stored in `argument`. At most `on` characters of the current string object starting at offset `opos` are compared to at most `an` characters of `argument`. `Argument` must have at least `an` characters. The characters may have arbitrary values: the ASCII-Z value has no special meaning.

- `size_t copy(char *argument, size_t on, size_type opos) const`:

  the contents of the current string object are (partially) copied into `argument`. The actual number of characters copied is returned. The second argument, specifying the number of characters to copy, from the current string object is required. No `ASCII-Z` is appended to the copied string but can be appended to the copied text using an idiom like the following:

  ```
  argument[object.copy(argument, string::npos)] = 0;
  ```

  Of course, the programmer should make sure that `argument`'s size is large enough to accomodate the additional 0-byte.

- `char const *c_str() const`:

  the contents of the current string object as an ASCII-Z terminated **C**-string.

- `char const *data() const`:

  the raw contents of the current string object are returned. Since this member does not return an ASCII-Z terminated **C** string (as `c_str` does), it can be used to retrieve any kind of information stored inside the current string object including, e.g., series of 0-bytes:

  ```
  string s(2, 0);
  cout << static_cast<int>(s.data()[1]) << '\n';
  ```

- `bool empty() const`:

  `true` is returned if the current string object contains no data.

- `string &erase(size_type opos, size_type on)`:

  a (sub)string of the information stored in the current string object is erased.

- `string::iterator erase(string::iterator begin, string::iterator end)`:

  the parameter `end` is optional. If omitted the value returned by the current object's `end` member is used. The characters defined by the `begin` and `end` iterators are erased. The iterator `begin` is returned, which is then referring to the position immediately following the last erased character.

- `size_t find(string const &argument, size_type opos) const`:

  the first index in the current string object where `argument` is found is returned.

- `size_t find(char const *argument, size_type opos, size_type an) const`:

  the first index in the current string object where `argument` is found is returned. When all three arguments are specified the first argument *must* be a `char const *`.

- `size_t find(char ch, size_type opos) const`:

  the first index in the current string object where `ch` is found is returned.

- `size_t find_first_of(string const &argument, size_type opos) const`:

  the first index in the current string object where any character in `argument` is found is returned.

- `size_type find_first_of(char const *argument, size_type opos, size_type an) const`:

  the first index in the current string object where any character in `argument` is found is returned. If `opos` is provided it refers to the first index in the current string object where the search for `argument` should start. If omitted, the string object is scanned completely. If `an` is provided it indicates the number of characters of the `char const *` argument that should be used in the search. It defines a substring starting at the beginning of `argument`. If omitted, all of `argument`'s characters are used.

- `size_type find_first_of(char ch, size_type opos)`:

  the first index in the current string object where character `ch` is found is returned.

- `size_t find_first_not_of(char ch, size_type opos) const`:

  the first index in the current string object where another character than `ch` is found is returned.

- `size_t find_last_of(string const &argument, size_type opos) const`:

  the last index in the current string object where any character in `argument` is found is returned.

- `size_type find_last_of(char const *argument, size_type opos, size_type an) const`:

  the last index in the current string object where any character in `argument` is found is returned. If `opos` is provided it refers to the last index in the current string object where the search for `argument` should start. If omitted, the string object is scanned completely. If `an` is provided it indicates the number of characters of the `char const *` argument that should be used in the search. It defines a substring starting at the beginning of `argument`. If omitted, all of `argument`'s characters are used.

- `size_type find_last_of(char ch, size_type opos)`:

  the last index in the current string object where character `ch` is found is returned.

- `size_t find_last_not_of(string const &argument, size_type opos) const`:

  the last index in the current string object where any character *not* appearing in `argument` is found is returned.

- `istream &std::getline(istream &istr, string &object, char delimiter = '\n')`:

  Note: this is *not* a member function of the class `string`.
  A line of text is read from `istr`. All characters until `delimiter` (or the end of the stream, whichever comes first) are read from `istr` and are stored in `object`. If the delimiter is encountered it is removed from the stream, but is not stored in `line`.
  If the delimiter is not found, `istr.eof` returns `true` (see section 6.3.1). Since streams may be interpreted as `bool` values (cf. section 6.3.1) a commonly encountered idiom to read all lines from a stream successively into a string object `line` looks like this:

  ```
  while (getline(istr, line))
      process(line);
  ```

  The contents of the last line, whether or not it was terminated by a delimiter, is eventually also assigned to `object`.

- `string &insert(size_t opos, string const &argument, size_type apos, size_type an)`

  a (sub)string of `argument` is inserted into the current string object at the current string object's index position `opos`. Arguments for `apos` and `an` must either both be provided or they must both be omitted.

- `string &insert(size_t opos, char const *argument, size_type an)`:

  `argument` (of type `char const *`) is inserted at index `opos` into the current string object.

- `string &insert(size_t opos, size_t count, char ch)`:

  `Count` characters `ch` are inserted at index `opos` into the current string object.

- `string::iterator insert(string::iterator begin, char ch)`:

  the character `ch` is inserted at the current object's position referred to by `begin`. `Begin` is returned.

- `string::iterator insert(string::iterator begin, size_t count, char ch)`:

  `Count` characters `ch` are inserted at the current object's position referred to by `begin`. `Begin` is returned.

- `string::iterator insert(string::iterator begin, InputIterator abegin, InputIterator aend)`:

  the characters in the range defined by the `InputIterators abegin` and `aend` are inserted at the current object's position referred to by `begin`. `Begin` is returned.

- `size_t length() const`:

  the number of characters stored in the current string object is returned.

- `size_t max_size() const`:

  the maximum number of characters that can be stored in the current string object is returned.

- `string &replace(size_t opos, size_t on, string const &argument, size_type apos, size_type an)`:

  a (sub)string of characters in `object` are replaced by the (subset of) characters of `argument`. If `on` is specified as 0 `argument` is inserted into `object` at offset `opos`.

- `string &replace(size_t opos, size_t on, char const *argument, size_type an)`:

  a series of characters in `object` are be replaced by the first `an` characters of `char const *` argument.

- `string &replace(size_t opos, size_t on, size_type count, char ch)`:

  `on` characters of the current string object, starting at index position `opos`, are replaced by `count` characters `ch`.

- `string &replace(string::iterator begin, string::iterator end, string const & argument)`:

  the series of characters in the current string object defined by the iterators `begin` and `end` are replaced by `argument`. If `argument` is a `char const *`, an additional argument `an` may be used, specifying the number of characters of `argument` that are used in the replacement.

- `string &replace(string::iterator begin, string::iterator end, size_type count, char ch)`:

  the series of characters in the current string object defined by the iterators `begin` and `end` are replaced by `count` characters having values `ch`.

- `string &replace(string::iterator begin, string::iterator end, InputIterator abegin, InputIterator aend)`:

  the series of characters in the current string object defined by the iterators `begin` and `end` are replaced by the characters in the range defined by the `InputIterators` `abegin` and `aend`.

- `void reserve(size_t request)`:

  the current string object's capacity is changed to at least `request`. After calling this member, `capacity`'s return value will be at least `request`. A request for a smaller size than the value returned by `capacity` is ignored. A `std::length_error` exception is thrown if `request` exceeds the value returned by `max_size` (`std::length_error` is defined in the `stdexcept` header). Calling `reserve()` has the effect of redefining a string's capacity, not of actually making available the memory to the program. This is illustrated by the exception thrown by the string's `at()` member when trying to access an element exceeding the string's `size` but not the string's `capacity`.

- `void resize(size_t size, char ch = 0)`:

  the current string object is resized to `size` characters. If the string object is resized to a size larger than its current size the additional characters will be initialized to `ch`. If it is reduced in size the characters having the highest indices are chopped off.

- `size_t rfind(string const &argument, size_type opos) const`:

  the last index in the current string object where `argument` is found is returend. Searching proceeds from the current object's offset `opos` back to its beginning.

- `size_t rfind(char const *argument, size_type opos, size_type an) const`:

  the last index in the current string object where `argument` is found is returned. Searching proceeds from the current object's offset `opos` back to its beginning. The parameter `an` specifies the length of the substring of `argument` to look for, starting at `argument`'s beginning.

- `size_t rfind(char ch, size_type opos)const`:

  the last index in the current string object where `ch` is found is returned. Searching proceeds from the current object's offset `opos` back to its beginning.

- `size_t size() const`:

  the number of characters stored in the current string object is returned. This member is a synonym of `length()`.

- `string substr(size_type opos, size_type on) const`:

  a substring of the current string object of at most `on` characters starting at index `opos` is returned.

- `void swap(string &argument)`:

  the contents of the current string object are swapped with the contents of `argument`. For this member `argument` must be a string object and cannot be a `char const *`.

Chapter 6

The IO-stream Library

Extending the standard stream (`FILE`) approach, well known from the **C** programming language, **C++** offers an *input*/*output* (I/O) library based on `class` concepts.

All **C++** I/O facilities are defined in the namespace `std`. The `std::` prefix is omitted below, except for situations where this would result in ambiguities.

Earlier (in chapter 3) we've seen several examples of the use of the **C++** I/O library, in particular showing insertion operator (<<) and the extraction operator (>>). In this chapter we'll cover I/O in more detail.

The discussion of input and output facilities provided by the **C++** programming language heavily uses the `class` concept and the notion of member functions. Although class construction has not yet been covered (for that see chapter 7) and although *inheritance* will not be covered formally before chapter 13, we think it is quite possible to introduce I/O facilities long before the technical background of their construction has been covered.

Most **C++** I/O classes have names starting with `basic_` (like `basic_ios`). However, these `basic_` names are not regularly found in **C++** programs, as most classes are also defined using `typedef` definitions like:

```
typedef basic_ios<char>       ios;
```

Since **C++** supports various kinds of character types (e.g., `char`, `wchar_t`), I/O facilities were developed using the *template* mechanism allowing for easy conversions to character types other than the traditional `char` type. As elaborated in chapter 20, this also allows the construction of generic software, that could thereupon be used for any particular type representing characters. So, analogously to the above `typedef` there exists a

```
typedef basic_ios<wchar_t>    wios;
```

This type definition can be used for the `wchar_t` type. Because of the existence of these type definitions, the `basic_` prefix was omitted from the **C++** Annotations without loss of continuity. The **C++** Annotations primarily focus on the standard 8-bits `char` type.

It must be stressed that it is *not* correct anymore to declare iostream objects using standard forward declarations, like:

```
class std::ostream;     // now erroneous
```

Instead, sources that must declare iostream classes must

```
#include <iosfwd>       // correct way to declare iostream classes
```

Using **C++** I/O offers the additional advantage of *type safety*. Objects (or plain values) are inserted into streams. Compare this to the situation commonly encountered in **C** where the `fprintf` function is used to indicate by a format string what kind of value to expect where. Compared to this latter situation **C++**'s *iostream* approach immediately uses the objects where their values should appear, as in

```
cout << "There were " << nMaidens << " virgins present\n";
```

The compiler notices the type of the `nMaidens` variable, inserting its proper value at the appropriate place in the sentence inserted into the `cout` iostream.

Compare this to the situation encountered in **C**. Although **C** compilers are getting smarter and smarter, and although a well-designed **C** compiler may warn you for a mismatch between a format specifier and the type of a variable encountered in the corresponding position of the argument list of a `printf` statement, it can't do much more than *warn* you. The *type safety* seen in **C++** *prevents* you from making type mismatches, as there are no types to match.

Apart from this, *iostreams* offer more or less the same set of possibilities as the standard `FILE`-based I/O used in **C**: files can be opened, closed, positioned, read, written, etc.. In **C++** the basic `FILE` structure, as used in **C**, is still available. But **C++** adds to this I/O based on classes, resulting in type safety, extensibility, and a clean design.

In the ANSI/ISO standard the intent was to create architecture independent I/O. Previous implementations of the iostreams library did not always comply with the standard, resulting in many extensions to the standard. The I/O sections of previously developed software may have to be partially rewritten. This is tough for those who are now forced to modify old software, but every feature and extension that was once available can be rebuilt easily using ANSI/ISO standard conforming I/O. Not all of these reimplementations can be covered in this chapter, as many reimplementations relies on inheritance and polymorphism, which will not be covered until chapters 13 and 14. Selected reimplementations will be provided in chapter 23, and in this chapter references to particular sections in other chapters will be given where appropriate. This chapter is organized as follows (see also Figure 6.1):

- The `class ios_base` is the foundation upon which the iostreams I/O library was built. It defines the core of all I/O operations and offers, among other things, facilities for inspecting the state of I/O streams and for output formatting.
- The class `ios` was directly derived from `ios_base`. Every class of the I/O library doing input or output is itself *derived* from this `ios` class, and so *inherits* its (and, by implication, `ios_base`'s) capabilities. The reader is urged to keep this in mind while reading this chapter. The concept of inheritance is not discussed here, but rather in chapter 13.

  The class `ios` is important in that it implements the communication with a *buffer* that is used by streams. This buffer is a `streambuf` object which is responsible for the actual I/O to/from the underlying *device*. Consequently `iostream` objects do not perform I/O operations themselves, but leave these to the (stream)buffer objects with which they are associated.
- Next, basic **C++** output facilities are discussed. The basic class used for output operations is `ostream`, defining the insertion operator as well as other facilities writing information to streams. Apart from inserting information into files it is possible to insert information into memory buffers, for which the `ostringstream` class is available. Formatting output is to a great extent possible using the facilities defined in the `ios` class, but it is also possible to *insert formatting commands* directly into streams using *manipulators*. This aspect of **C++** output is discussed as well.
- Basic **C++** input facilities are implemented by the `istream` class. This class defines the extraction operator and related input facilities. Comparably to insert information into to memory buffers (using `ostringstream`) a class `istringstream` is available to extract information from memory buffers.
- Finally, several advanced I/O-related topics are discussed. E.g., reading and writing from the same stream and mixing **C** and **C++** I/O using `filebuf` ojects. Other I/O related topics are covered elsewhere in the **C++** Annotations, e.g., in chapter 23.

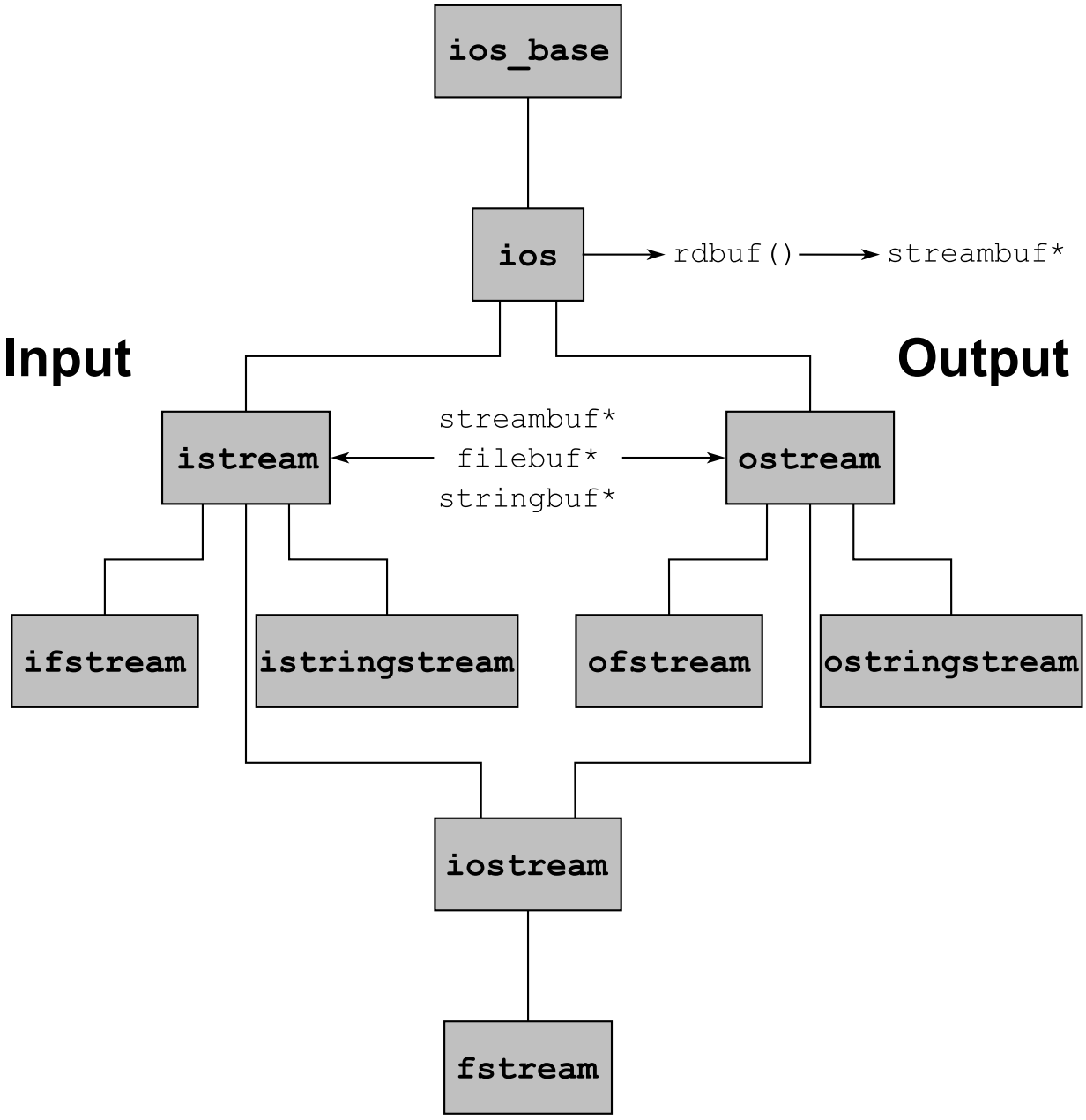

Figure 6.1: Central I/O Classes

Stream objects have a limited but important role: they are the interface between, on the one hand, the objects to be input or output and, on the other hand, the `streambuf`, which is responsible for the actual input and output to the device accessed by a `streambuf` object.

This approach allows us to construct a new kind of `streambuf` for a new kind of device, and use that streambuf in combination with the 'good old' `istream`- and `ostream`-class facilities. It is important to understand the distinction between the formatting roles of iostream objects and the buffering interface to an external device as implemented in a `streambuf` object. Interfacing to new devices (like *sockets* or *file descriptors*) requires the construcion of a new kind of `streambuf`, rather than a new kind of `istream` or `ostream` object. A *wrapper class* may be constructed around the `istream` or `ostream` classes, though, to ease the access to a special device. This is how the stringstream classes were constructed.

6.1 Special header files

Several iostream related header files are available. Depending on the situation at hand, the following header files should be used:

- `iosfwd`: sources should include this header file if only a declaration of the stream classes is required. For example, if a function defines a reference parameter to an `ostream` then the compiler doesn not need to know exactly what an `ostream` is. When declaring such a function the `ostream` class merely needs to be be declared. One cannot use

  ```
  class std::ostream; // erroneous declaration

  void someFunction(std::ostream &str);
  ```

  but, instead, one should use:

  ```
  #include <iosfwd>   // correctly declares class ostream

  void someFunction(std::ostream &str);
  ```

- `<streambuf>`: sources should include this header file when using `streambuf` or `filebuf` classes. See sections 14.7 and 14.7.2.
- `<istream>`: sources should include this preprocessor directive when using the class `istream` or when using classes that do both input and output. See section 6.5.1.
- `<ostream>`: sources should include this header file when using the class `ostream` class or when using classes that do both input and output. See section 6.4.1.
- `<iostream>`: sources should include this header file when using the global stream objects (like `cin` and `cout`).
- `<fstream>`: sources should include this header file when using the file stream classes. See sections 6.4.2, 6.5.2, and 6.6.2.
- `<sstream>`: sources should include this header file when using the string stream classes. See sections 6.4.3 and 6.5.3.
- `<iomanip>`: sources should include this header file when using parameterized manipulators. See section 6.3.2.

6.2 The foundation: the class 'ios_base'

The class `std::ios_base` forms the foundation of all I/O operations, and defines, among other things, facilities for inspecting the state of I/O streams and most output formatting facilities. Every stream

class of the I/O library is, through the class `ios`, *derived* from this class, and *inherits* its capabilities. As `ios_base` is the foundation on which all **C++** I/O was built, we introduce it here as the first class of the **C++** I/O library.

Note that, as in **C**, I/O in **C++** is *not* part of the language (although it *is* part of the ANSI/ISO standard on **C++**). Although it is technically possible to ignore all predefined I/O facilities, nobody does so, and the I/O library therefore represents a *de facto* I/O standard for **C++**. Also note that, as mentioned before, the iostream classes themselves are not responsible for the eventual I/O, but delegate this to an auxiliary class: the class `streambuf` or its derivatives.

It is neither possible nor required to construct an `ios_base` object directly. Its construction is always a side-effect of constructing an object further down the class hierarchy, like `std::ios`. `Ios` is the next class down the iostream hierarchy (see figure 6.1). Since all stream classes in turn inherit from `ios`, and thus also from `ios_base`, the distinction between `ios_base` and `ios` is in practice not important. Therefore, facilities actually provided by `ios_base` will be discussed as facilities provided by `ios`. The reader who is interested in the true class in which a particular facility is defined should consult the relevant header files (e.g., `ios_base.h` and `basic_ios.h`).

6.3 Interfacing 'streambuf' objects: the class 'ios'

The `std::ios` class is derived directly from `ios_base`, and it defines *de facto* the foundation for all stream classes of the **C++** I/O library.

Although it *is* possible to construct an `ios` object directly, this is seldom done. The purpose of the class `ios` is to provide the facilities of the class `basic_ios`, and to add several new facilites, all related to the `streambuf` object which is managed by objects of the class `ios`.

All other stream classes are either directly or indirectly derived from `ios`. This implies, as explained in chapter 13, that all facilities of the classes `ios` and `ios_base` are also available to other stream classes. Before discussing these additional stream classes, the features offered by the class `ios` (and by implication: by `ios_base`) are now introduced.

In some cases it may be required to include `ios` explicitly. An example is the situations where the formatting flags themselves (cf. section 6.3.2.2) are referred to in source code.

The class `ios` offers several member functions, most of which are related to formatting. Other frequently used member functions are:

- `std::streambuf *ios::rdbuf()`:

  A pointer to the `streambuf` object forming the interface between the `ios` object and the device with which the `ios` object communicates is returned. See section 23.2.2 for further information about the `class streambuf`.

- `std::streambuf *ios::rdbuf(std::streambuf *new)`:

  The current `ios` object is associated with another `streambuf` object. A pointer to the `ios` object's original `streambuf` object is returned. The object to which this pointer points is not destroyed when the `stream` object goes out of scope, but is owned by the caller of `rdbuf`.

- `std::ostream *ios::tie()`:

  A pointer to the `ostream` object that is currently tied to the `ios` object is returned (see the next member). The return value 0 indicates that currently no `ostream` object is tied to the `ios` object. See section 6.5.5 for details.

- `std::ostream *ios::tie(std::ostream *outs)`:

  The ostream object is tied to current `ios` object. This means that the `ostream` object is *flushed* every time before an input or output action is performed by the current `ios`

object. A pointer to the `ios` object's original `ostream` object is returned. To break the tie, pass the argument 0. See section 6.5.5 for an example.

6.3.1 Condition states

Operations on streams may fail for various reasons. Whenever an operation fails, further operations on the stream are suspended. It is possible to inspect (and possibly: clear) the condition state of streams, allowing a program to repair the problem rather than having to abort.

Conditions are represented by the following *condition flags*:

- `ios::badbit`:

  if this flag has been raised an illegal operation has been requested at the level of the `streambuf` object to which the stream interfaces. See the member functions below for some examples.

- `ios::eofbit`:

  if this flag has been raised, the `ios` object has sensed end of file.

- `ios::failbit`:

  if this flag has been raised, an operation performed by the stream object has failed (like an attempt to extract an `int` when no numeric characters are available on input). In this case the stream itself could not perform the operation that was requested of it.

- `ios::goodbit`:

  this flag is raised when none of the other three condition flags were raised.

Several condition member functions are available to manipulate or determine the states of `ios` objects. Originally they returned `int` values, but their current return type is `bool`:

- `ios::bad()`:

  the value `true` is returned when the stream's `badbit` has been set and `false` otherwise. If `true` is returned it indicates that an illegal operation has been requested at the level of the `streambuf` object to which the stream interfaces. What does this mean? It indicates that the `streambuf` itself is behaving unexpectedly. Consider the following example:

  ```
  std::ostream error(0);
  ```

  Here an `ostream` object is constructed *without* providing it with a working `streambuf` object. Since this '`streambuf`' will never operate properly, its `badbit` flag is raised from the very beginning: `error.bad()` returns `true`.

- `ios::eof()`:

  the value `true` is returned when end of file (`EOF`) has been sensed (i.e., the `eofbit` flag has been set) and `false` otherwise. Assume we're reading lines line-by-line from `cin`, but the last line is not terminated by a final `\n` character. In that case `std::getline` attempting to read the `\n` delimiter hits end-of-file first. This raises the `eofbit` flag and `cin.eof()` returns `true`. For example, assume `std::string str` and `main` executing the statements:

  ```
  getline(cin, str);
  cout << cin.eof();
  ```

  Then

  ```
  echo "hello world" | program
  ```

prints the value 0 (no EOF sensed). But after

```
echo -n "hello world" | program
```

the value 1 (EOF sensed) is printed.

- `ios::fail()`:

  the value `true` is returned when `bad` returns `true` or when the `failbit` flag was set. The value `false` is returned otherwise. In the above example, `cin.fail()` returns `false`, whether we terminate the final line with a delimiter or not (as we've read a line). However, trying to execute a second `getline` will set the `failbit` flag, causing `cin::fail()` to return `true`. In general: `fail` returns `true` if the requested stream operation failed. A simple example showing this consists of an attempt to extract an `int` when the input stream contains the text `hello world`. The value `not fail()` is returned by the `bool` interpretation of a stream object (see below).

- `ios::good()`:

  the value of the `goodbit` flag is returned. It equals `true` when none of the other condition flags (`badbit, eofbit, failbit`) was raised. Consider the following little program:

```
#include <iostream>
#include <string>

using namespace std;

void state()
{
    cout << "\n"
            "Bad: " << cin.bad() << " "
            "Fail: " << cin.fail() << " "
            "Eof: " << cin.eof() << " "
            "Good: " << cin.good() << '\n';
}

int main()
{
    string line;
    int x;

    cin >> x;
    state();

    cin.clear();
    getline(cin, line);
    state();

    getline(cin, line);
    state();
}
```

  When this program processes a file having two lines, containing, respectively, `hello` and `world`, while the second line is not terminated by a `\n` character the following is shown:

```
Bad: 0 Fail: 1 Eof: 0 Good: 0

Bad: 0 Fail: 0 Eof: 0 Good: 1

Bad: 0 Fail: 0 Eof: 1 Good: 0
```

Thus, extracting x fails (good returning false). Then, the error state is cleared, and the first line is successfully read (good returning true). Finally the second line is read (incompletely): good returning false, and eof returning true.

- Interpreting streams as bool values:

  streams may be used in expressions expecting logical values. Some examples are:

  ```
  if (cin)                // cin itself interpreted as bool
  if (cin >> x)           // cin interpreted as bool after an extraction
  if (getline(cin, str))  // getline returning cin
  ```

  When interpreting a stream as a logical value, it is actually 'not fail()' that is interpreted. The above examples may therefore be rewritten as:

  ```
  if (not cin.fail())
  if (not (cin >> x).fail())
  if (not getline(cin, str).fail())
  ```

  The former incantation, however, is used almost exclusively.

The following members are available to manage error states:

- ios::clear():

  When an error condition has occurred, and the condition can be repaired, then clear can be used to clear the error state of the file. An overloaded version exists accepting state flags, that are set after first clearing the current set of flags: clear(int state). Its return type is void

- ios::rdstate():

  The current set of flags that are set for an ios object are returned (as an int). To test for a particular flag, use the bitwise and operator:

  ```
  if (!(iosObject.rdstate() & ios::failbit))
  {
      // last operation didn't fail
  }
  ```

  Note that this test cannot be performed for the goodbit flag as its value equals zero. To test for 'good' use a construction like:

  ```
  if (iosObject.rdstate() == ios::goodbit)
  {
      // state is 'good'
  }
  ```

- ios::setstate(ios::iostate state):

  A stream may be assigned a certain set of states using setstate. Its return type is void. E.g.,

  ```
  cin.setstate(ios::failbit);     // set state to 'fail'
  ```

  To set multiple flags in one setstate() call use the bitor operator:

  ```
  cin.setstate(ios::failbit | ios::eofbit)
  ```

  The member clear is a shortcut to clear all error flags. Of course, clearing the flags doesn't automatically mean the error condition has been cleared too. The strategy should be:

  - An error condition is detected,
  - The error is repaired
  - The member clear is called.

C++ supports an *exception* mechanism to handle exceptional situations. According to the ANSI/ISO standard, exceptions can be used with stream objects. Exceptions are covered in chapter 9. Using exceptions with stream objects is covered in section 9.7.

6.3.2 Formatting output and input

The way information is written to streams (or, occasionally, read from streams) is controlled by *formatting flags*.

Formatting is used when it is necessary to, e.g., set the width of an output field or input buffer and to determine the form (e.g., the *radix*) in which values are displayed. Most formatting features belong to the realm of the `ios` class. Formatting is controlled by flags, defined by the `ios` class. These flags may be manipulated in two ways: using specialized member functions or using *manipulators*, which are directly inserted into or extracted from streams. There is no special reason for using either method; usually both methods are possible. In the following overview the various member functions are first introduced. Following this the flags and manipulators themselves are covered. Examples are provided showing how the flags can be manipulated and what their effects are.

Many manipulators are parameterless and are available once a stream header file (e.g., `iostream`) has been included. Some manipulators require arguments. To use the latter manipulators the header file `iomanip` must be included.

6.3.2.1 Format modifying member functions

Several *member functions* are available manipulating the I/O formatting flags. Instead of using the members listed below *manipulators* are often available that may directly be inserted into or extracted from streams. The available members are listed in alphabetical order, but the most important ones in practice are setf, unsetf and width.

- `ios &ios::copyfmt(ios &obj)`:

  all format flags of `obj` are copied to the current `ios` object. The current `ios` object is returned.

- `ios::fill() const`:

  the current padding character is returned. By default, this is the blank space.

- `ios::fill(char padding)`:

  the padding character is redefined, the *previous* padding character is returned. Instead of using this member function the `setfill` *manipulator* may be inserted directly into an `ostream`. Example:

```
cout.fill('0');          // use '0' as padding char
cout << setfill('+');    // use '+' as padding char
```

- `ios::fmtflags ios::flags() const`:

  the current set of flags controlling the format state of the stream for which the member function is called is returned. To inspect whether a particular flag was set, use the `bit_and` operator. Example:

```
if (cout.flags() & ios::hex)
    cout << "Integral values are printed as hex numbers\n"
```

- `ios::fmtflags ios::flags(ios::fmtflags flagset)`:

  the *previous* set of flags are returned and the new set of flags are defined by `flagset`. Multiple flags are specified using the `bitor` operator. Example:

```
    // change the representation to hexadecimal
cout.flags(ios::hex | cout.flags() & ~ios::dec);
```

- `int ios::precision() const`:

  the number of significant digits used when outputting floating point values is returned (default: 6).

- `int ios::precision(int signif)`:

  the number of significant digits to use when outputting real values is set to `signif`. The previously used number of significant digits is returned. If the number of required digits exceeds `signif` then the number is displayed in 'scientific' notation (cf. section 6.3.2.2). Manipulator: `setprecision`. Example:

```
cout.precision(3);          // 3 digits precision
cout << setprecision(3);    // same, using the manipulator

cout << 1.23 << " " << 12.3 << " " << 123.12 << " " << 1234.3 << endl;
// displays: 1.23 12.3 123 1.23e+03
```

- `ios::fmtflags ios::setf(ios::fmtflags flags)`:

  sets one or more formatting flags (use the `bitor` operator to combine multiple flags). Already set flags are not affected. The *previous* set of flags is returned. Instead of using this member function the manipulator `setiosflags` may be used. Examples are provided in the next section (6.3.2.2).

- `ios::fmtflags ios::setf(ios::fmtflags flags, ios::fmtflags mask)`:

  clears all flags mentioned in `mask` and sets the flags specified in `flags`. The *previous* set of flags is returned. Some examples are (but see the next section (6.3.2.2) for a more thorough discussion):

```
    // left-adjust information in wide fields
cout.setf(ios::left, ios::adjustfield);
    // display integral values as hexadecimal numbers
cout.setf(ios::hex, ios::basefield);
    // display floating point values in scientific notation
cout.setf(ios::scientific, ios::floatfield);
```

- `ios::fmtflags ios::unsetf(fmtflags flags)`:

  the specified formatting flags are cleared (leaving the remaining flags unaltered) and returns the *previous* set of flags. A request to unset an active default flag (e.g., `cout.unsetf(ios::dec)`) is ignored. Instead of this member function the manipulator `resetiosflags` may also be used. Example:

```
cout << 12.24;              // displays  12.24
cout << setf(ios::fixed);
cout << 12.24;              // displays  12.240000
cout.unsetf(ios::fixed);    // undo a previous ios::fixed setting.
cout << 12.24;              // displays  12.24
cout << resetiosflags(ios::fixed);  // using manipulator rather
                                    // than unsetf
```

- `int ios::width() const`:

  the currently active output field width to use on the next insertion is returned. The default value is 0, meaning 'as many characters as needed to write the value'.

- `int ios::width(int nchars)`:

  the field width of the next insertion operation is set to `nchars`, returning the previously used field width. This setting is not persistent. It is reset to 0 after every insertion operation. Manipulator: `std::setw(int)`. Example:

```
cout.width(5);
cout << 12;                     // using 5 chars field width
cout << setw(12) << "hello";    // using 12 chars field width
```

6.3.2.2 Formatting flags

Most *formatting flags* are related to outputting information. Information can be written to output streams in basically two ways: using *binary output* information is written directly to an output stream, without converting it first to some human-readable format and using *formatted output* by which values stored in the computer's memory are converted to human-readable text first. Formatting flags are used to define the way this conversion takes place. In this section all formatting flags are covered. Formatting flags may be (un)set using member functions, but often manipulators having the same effect may also be used. For each of the flags it is shown how they can be controlled by a member function or -if available- a manipulator.

To display information in wide fields:

- `ios::internal`:

  to add fill characters (blanks by default) between the minus sign of negative numbers and the value itself. Other values and data types are right-adjusted. Manipulator: `std::internal`. Example:

```
cout.setf(ios::internal, ios::adjustfield);
cout << internal;       // same, using the manipulator

cout << '\'' << setw(5) << -5 << "'\n";  // displays '-   5'
```

- `ios::left`:

  to left-adjust values in fields that are wider than needed to display the values. Manipulator: `std::left`. Example:

```
cout.setf(ios::left, ios::adjustfield);
cout << left;           // same, using the manipulator

cout << '\'' << setw(5) << "hi" << "'\n";  // displays 'hi   '
```

- `ios::right`:

  to right-adjust values in fields that are wider than needed to display the values. Manipulator: `std::right`. This is the default. Example:

```
cout.setf(ios::right, ios::adjustfield);
cout << right;          // same, using the manipulator

cout << '\'' << setw(5) << "hi" << "'\n";  // displays '   hi'
```

Using various number representations:

- `ios::dec`:

  to display integral values as decimal numbers. Manipulator: `std::dec`. This is the default. Example:

```
cout.setf(ios::dec, ios::basefield);
cout << dec;            // same, using the manipulator
cout << 0x10;           // displays 16
```

- `ios::hex`:

  to display integral values as hexadecimal numbers. Manipulator: `std::hex`. Example:

```
cout.setf(ios::hex, ios::basefield);
cout << hex;            // same, using the manipulator
cout << 16;             // displays 10
```

- `ios::oct`:

  to display integral values as octal numbers. Manipulator: `std::oct`. Example:

```
cout.setf(ios::oct, ios::basefield);
cout << oct;            // same, using the manipulator
cout << 16;             // displays 20
```

- `std::setbase(int radix)`:

  This is a manipulator that can be used to change the number representation to decimal, hexadecimal or octal. Example:

```
cout << setbase(8);     // octal numbers, use 10 for
                        // decimal, 16 for hexadecimal
cout << 16;             // displays 20
```

Fine-tuning displaying values:

- `ios::boolalpha`:

  logical values may be displayed as text using the text 'true' for the `true` logical value, and 'false' for the `false` logical value using `boolalpha`. By default this flag is not set. Complementary flag: `ios::noboolalpha`. Manipulators: `std::boolalpha` and `std::noboolalpha`. Example:

```
cout.setf(ios::boolalpha);
cout << boolalpha;      // same, using the manipulator
cout << (1 == 1);       // displays true
```

- `ios::showbase`:

  to display the numeric base of integral values. With hexadecimal values the `0x` prefix is used, with octal values the prefix `0`. For the (default) decimal value no particular prefix is used. Complementary flag: `ios::noshowbase`. Manipulators: `std::showbase` and `std::noshowbase`. Example:

```
cout.setf(ios::showbase);
cout << showbase;       // same, using the manipulator
cout << hex << 16;      // displays 0x10
```

- `ios::showpos`:

  to display the + sign with positive decimal (only) values. Complementary flag: `ios::noshowpos`. Manipulators: `std::showpos` and `std::noshowpos`. Example:

```
cout.setf(ios::showpos);
cout << showpos;            // same, using the manipulator
cout << 16;                 // displays +16
cout.unsetf(ios::showpos);  // Undo showpos
cout << 16;                 // displays 16
```

- `ios::uppercase`:

  to display letters in hexadecimal values using capital letters. Complementary flag: `ios::nouppercase`. Manipulators: `std::uppercase` and `std::nouppercase`. By default lower case letters are used. Example:

```
cout.setf(ios::uppercase);
cout << uppercase;          // same, using the manipulator
cout << hex << showbase <<
        3735928559;         // displays 0XDEADBEEF
```

Displaying floating point numbers

- `ios::fixed`:

  to display real values using a fixed decimal point (e.g., 12.25 rather than 1.225e+01), the `fixed` formatting flag is used. It can be used to set a fixed number of digits behind the decimal point. Manipulator: `fixed`. Example:

```
cout.setf(ios::fixed, ios::floatfield);
cout.precision(3);          // 3 digits behind the .

    // Alternatively:
cout << setiosflags(ios::fixed) << setprecision(3);

cout << 3.0 << " " << 3.01 << " " << 3.001 << '\n';
     << 3.0004 << " " << 3.0005 << " " << 3.0006 << '\n'
    // Results in:
    // 3.000 3.010 3.001
    // 3.000 3.001 3.001
```

  The example shows that 3.0005 is rounded away from zero, becoming 3.001 (likewise -3.0005 becomes -3.001). First setting precision and then `fixed` has the same effect.

- `ios::scientific`:

  to display real values in *scientific notation* (e.g., 1.24e+03). Manipulator: `std::scientific`. Example:

```
cout.setf(ios::scientific, ios::floatfield);
cout << scientific;         // same, using the manipulator
cout << 12.25;              // displays 1.22500e+01
```

- `ios::showpoint`:

  to display a trailing decimal point *and* trailing decimal zeros when real numbers are displayed. Complementary flag: `ios::noshowpoint`. Manipulators: `std::showpoint`, `std::noshowpoint`. Example:

```
cout << fixed << setprecision(3);   // 3 digits behind .

cout.setf(ios::showpoint);      // set the flag
cout << showpoint;              // same, using the manipulator

cout << 16.0 << ", " << 16.1 << ", " << 16;
// displays: 16.0000, 16.1000, 16
```

  Note that the final 16 is an integral rather than a floating point number, so it has no decimal point. So `showpoint` has no effect. If `ios::showpoint` is not active trailing zeros are discarded. If the fraction is zero the decimal point is discarded as well. Example:

```
cout.unsetf(ios::fixed, ios::showpoint);    // unset the flags

cout << 16.0 << ", " << 16.1;
// displays: 16, 16.1
```

Handling white space and flushing streams

- `std::endl`:

  manipulator inserting a newline character and flushing the stream. Often flushing the stream is not required and doing so would needlessly slow down I/O processing. Consequently, using `endl` should be avoided (in favor of inserting `'\n'`) unless flusing the stream is explicitly intended. Note that streams are automatically flushed when the

program terminates or when a stream is 'tied' to another stream (cf. `tie` in section 6.3). Example:

```
cout << "hello" << endl;    // prefer: << '\n';
```

- `std::ends`:

  manipulator inserting a 0-byte into a stream. It is usually used in combination with memory-streams (cf. section 6.4.3).

- `std::flush`:

  a stream may be flushed using this member. Often flushing the stream is not required and doing so would needlessly slow down I/O processing. Consequently, using `flush` should be avoided unless it is explicitly required to do so. Note that streams are automatically flushed when the program terminates or when a stream is 'tied' to another stream (cf. `tie` in section 6.3). Example:

  ```
  cout << "hello" << flush;    // avoid if possible.
  ```

- `ios::skipws`:

  leading white space characters (blanks, tabs, newlines, etc.) are skipped when a value is extracted from a stream. This is the default. If the flag is not set, leading white space characters are not skipped. Complementary flag: `ios::noskipws`. Manipulators: `std::skipws`, `std::noskipws`. Example:

  ```
  cin.setw(ios::skipws);
  cin >> skipws;      // same, using the manipulator
  int value;
  cin >> value;       // skips initial blanks
  ```

- `ios::unitbuf`:

  the stream for which this flag is set will flush its buffer after every output operation Often flushing a stream is not required and doing so would needlessly slow down I/O processing. Consequently, setting `unitbuf` should be avoided unless flusing the stream is explicitly intended. Note that streams are automatically flushed when the program terminates or when a stream is 'tied' to another stream (cf. `tie` in section 6.3). Complementary flag: `ios::nounitbuf`. Manipulators: `std::unitbuf`, `std::nounitbuf`. Example:

  ```
  cout.setf(ios::unitbuf);
  cout << unitbuf;            // same, using the manipulator

  cout.write("xyz", 3);       // flush follows write.
  ```

- `std::ws`:

  manipulator removing all white space characters (blanks, tabs, newlines, etc.) at the current file position. White space are removed if present even if the flag `ios::noskipws` has been set. Example (assume the input contains 4 blank characters followed by the character `X`):

  ```
  cin >> ws;      // skip white space
  cin.get();      // returns 'X'
  ```

6.4 Output

In **C++** output is primarily based on the `std::ostream` class. The `ostream` class defines the basic operators and members inserting information into streams: the *insertion operator* (`<<`), and special members like `write` writing unformatted information to streams.

The class `ostream` acts as *base class* for several other classes, all offering the functionality of the `ostream` class, but adding their own specialties. In the upcoming sections we will introduce:

- The class `ostream`, offering the basic output facilities;
- The class `ofstream`, allowing us to write files (comparable to **C**'s `fopen(filename, "w")`);
- The class `ostringstream`, allowing us to write information to memory (comparable to **C**'s `sprintf` function).

6.4.1 Basic output: the class 'ostream'

The class `ostream` defines basic output facilities. The `cout`, `clog` and `cerr` objects are all `ostream` objects. All facilities related to output as defined by the `ios` class are also available in the `ostream` class.

We may define `ostream` objects using the following *ostream constructor*:

- `std::ostream object(std::streambuf *sb)`:

  this constructor creates an `ostream` object which is a wrapper around an existing `std::streambuf` object. It isn't possible to define a plain `ostream` object (e.g., using `std::ostream out;`) that can thereupon be used for insertions. When `cout` or its friends are used, we are actually using a predefined `ostream` object that has already been defined for us and interfaces to the standard output stream using a (also predefined) `streambuf` object handling the actual interfacing.

  It *is*, however, possible to define an `ostream` object passing it a 0-pointer. Such an object cannot be used for insertions (i.e., it will raise its `ios::bad` flag when something is inserted into it), but it may be given a `streambuf` later. Thus it may be preliminary constructed, suspending its use until an appropriate `streambuf` becomes available.

To define the `ostream` class in **C++** sources, the `ostream` header file must be included. To use the predefined `ostream` objects (`std::cin, std::cout` etc.) the `iostream` header must be included.

6.4.1.1 Writing to 'ostream' objects

The class `ostream` supports both formatted and *binary output*.

The *insertion operator* (<<) is used to insert values in a type safe way into `ostream` objects. This is called formatted output, as binary values which are stored in the computer's memory are converted to human-readable ASCII characters according to certain formatting rules.

The insertion operator points to the `ostream` object to receive the information. The normal associativity of << remains unaltered, so when a statement like

```
cout << "hello " << "world";
```

is encountered, the leftmost two operands are evaluated first (`cout << "hello "`), and an `ostream &` object, which is actually the same `cout` object, is returned. Now, the statement is reduced to

```
cout << "world";
```

and the second string is inserted into `cout`.

The << operator has a lot of (overloaded) variants, so many types of variables can be inserted into `ostream` objects. There is an overloaded <<-operator expecting an `int`, a `double`, a pointer, etc. etc..

Each operator will return the `ostream` object into which the information so far has been inserted, followed by the next insertion.

Streams lack facilities for formatted output like **C**'s `printf` and `vprintf` functions. Although it is not difficult to implement these facilities in the world of streams, `printf`-like functionality is hardly ever required in **C++** programs. Furthermore, as it is potentially type-*unsafe*, it might be better to avoid this functionality completely.

When binary files must be written, normally no text-formatting is used or required: an `int` value should be written as a series of raw bytes, not as a series of ASCII numeric characters 0 to 9. The following member functions of `ostream` objects may be used to write 'binary files':

- `ostream& ostream::put(char c)`:

  to write a single character to the output stream. Since a character is a byte, this member function could also be used for writing a single character to a text-file.

- `ostream& ostream::write(char const *buffer, int length)`:

  to write at most `length` bytes, stored in the `char const *buffer` to the `ostream` object. Bytes are written as they are stored in the buffer, no formatting is done whatsoever. Note that the first argument is a `char const *`: a *type cast* is required to write any other type. For example, to write an `int` as an unformatted series of byte-values use:

  ```
  int x;
  out.write(reinterpret_cast<char const *>(&x), sizeof(int));
  ```

  The bytes written by the above `write` call will be written in an order depending on the *endianness* of the underlying hardware. Big-endian computers will write the most significant byte first, little-endian computers will first write the least significant byte.

6.4.1.2 'ostream' positioning

Although not every `ostream` object supports repositioning, they usually do. This means that it is possible to rewrite a section of the stream which was written earlier. Repositioning is frequently used in database applications where it must be possible to access the information in the database at random.

The current position can be obtained and modified using the following members:

- `ios::pos_type ostream::tellp()`:

  the current (absolute) position in the file where the next write-operation to the stream will take place is returned.

- `ostream &ostream::seekp(ios::off_type step, ios::seekdir org)`:

  modifies a stream's actual position. The function expects an `off_type step` representing the number of bytes the current stream position is moved with respect to `org`. The `step` value may be negative, zero or positive.
  The origin of the step, `org` is a value in the `ios::seekdir` enumeration. Its values are:
  - `ios::beg`:
    the stepsize is computed relative to the beginning of the stream. This value is used by default.
  - `ios::cur`:
    the stepsize is computed relative to the current position of the stream (as returned by `tellp`).
  - `ios::end`:
    the stepsize is interpreted relative to the current end position of the stream.

  It is OK to seek or write beyond the last file position. Writing bytes to a location beyond `EOF` will pad the intermediate bytes with ASCII-Z values: null-bytes. Seeking before `ios::beg` raises the `ios::fail` flag.

6.4.1.3 'ostream' flushing

Unless the `ios::unitbuf` flag has been set, information written to an `ostream` object is not immediately written to the physical stream. Rather, an internal buffer is filled during the write-operations, and when full it is flushed.

The stream's internal buffer can be flushed under program control:

- `ostream& ostream::flush()`:

  any buffered information stored internally by the `ostream` object is flushed to the device to which the `ostream` object interfaces. A stream is flushed automatically when:
  - the object ceases to exist;
  - the `endl` or `flush` *manipulators* (see section 6.3.2.2) are inserted into an `ostream` object;
  - a stream supporting the `close`-operation is explicitly closed (e.g., a `std::ofstream` object, cf. section 6.4.2).

6.4.2 Output to files: the class 'ofstream'

The `std::ofstream` class is derived from the `ostream` class: it has the same capabilities as the `ostream` class, but can be used to access files or create files for writing.

In order to use the `ofstream` class in **C++** sources, the `fstream` header file must be included. Including `fstream` will not automatically make available the standard streams `cin`, `cout` and `cerr`. Include `iostream` to declare these standard streams.

The following constructors are available for `ofstream` objects:

- `ofstream object`:

  this is the basic constructor. It defines an `ofstream` object which may be associated with an actual file later, using its `open()` member (see below).

- `ofstream object(char const *name, ios::openmode mode = ios::out)`:

  this constructor defines an `ofstream` object and associates it immediately with the file named `name` using output mode `mode`. Section 6.4.2.1 provides an overview of available output modes. Example:

  ```
  ofstream out("/tmp/scratch");
  ```

It is not possible to open an `ofstream` using a *file descriptor*. The reason for this is (apparently) that file descriptors are not universally available over different operating systems. Fortunately, file descriptors can be used (indirectly) with a `std::streambuf` object (and in some implementations: with a `std::filebuf` object, which is also a `streambuf`). `Streambuf` objects are discussed in section 14.7, `filebuf` objects are discussed in section 14.7.2.

Instead of directly associating an `ofstream` object with a file, the object can be constructed first, and opened later.

- `void ofstream::open(char const *name, ios::openmode mode = ios::out)`:

  this member function is used to associate an `ofstream` object with an actual file. If the `ios::fail` flag was set before calling `open` and opening succeeds the flag is cleared. Opening an already open stream fails. To reassociate a stream with another file it must first be closed:

  ```
  ofstream out("/tmp/out");
  ```

```
out << "hello\n";
out.close();        // flushes and closes out
out.open("/tmp/out2");
out << "world\n";
```

- `void ofstream::close()`:

  an `ofstream` object is closed by this member function. The function sets the `ios::fail` flag of the closed object. Closing the file will flush any buffered information to the associated file. A file is automatically closed when the associated `ofstream` object ceases to exist.

- `bool ofstream::is_open() const`:

  assume a stream was properly constructed, but it has not yet been attached to a file. E.g., the statement `ofstream ostr` was executed. When we now check its status through `good()`, a non-zero (i.e., *OK*) value is returned. The 'good' status here indicates that the stream object has been constructed properly. It doesn't mean the file is also open. To test whether a stream is actually open, `is_open` should be called. If it returns `true`, the stream is open. Example:

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ofstream of;

    cout << "of's open state: " << boolalpha << of.is_open() << '\n';

    of.open("/dev/null");       // on Unix systems

    cout << "of's open state: " << of.is_open() << '\n';
}
/*
    Generated output:
of's open state: false
of's open state: true
*/
```

6.4.2.1 Modes for opening stream objects

The following file modes or file flags are available when constructing or opening `ofstream` (or `istream`, see section 6.5.2) objects. The values are of type `ios::openmode`. Flags may be combined using the `bitor` operator.

- `ios::app`:

  reposition the stream to its end before every output command (see also `ios::ate` below). The file is created if it doesn't yet exist. When opening a stream in this mode any existing contents of the file are kept.

- `ios::ate`:

  start initially at the end of the file. Note that any existing contents are *only* kept if some other flag tells the object to do so. For example `ofstream out("gone", ios::ate)` will *rewrite* the file `gone`, because the implied `ios::out` will cause the rewriting. If rewriting of an existing file should be prevented, the `ios::in` mode should be specified too. However, when `ios::in` is specified the file must already exist. The `ate` mode

only initially positions the file at the end of file position. After that information may be written in the middle of the file using `seekp`. When the `app` mode is used information will *only* to written at end of file (effectively ignoring `seekp` operations).

- `ios::binary`:

  open a file in binary mode (used on systems distinguishing text- and binary files, like MS-Windows).

- `ios::in`:

  open the file for reading. The file must exist.

- `ios::out`:

  open the file for writing. Create it if it doesn't yet exist. If it exists, the file is rewritten.

- `ios::trunc`:

  start initially with an empty file. Any existing contents of the file are lost.

The following combinations of file flags have special meanings:

```
in | out:           The stream may be read and written. However, the
                    file must exist.
in | out | trunc:   The stream may be read and written. It is
                    (re)created empty first.
```

An interesting subtlety is that the `open` members of the `ifstream, ofstream` and `fstream` classes have a second parameter of type `ios::openmode`. In contrast to this, the `bitor` operator returns an `int` when applied to two enum-values. The question why the `bitor` operator may nevertheless be used here is answered in a later chapter (cf. section 10.11).

6.4.3 Output to memory: the class 'ostringstream'

To write information to memory using `stream` facilities, `std::ostringstream` objects should be used. As the class `ostringstream` is derived from the class `ostream` all `ostream`'s facilities are available to `ostringstream` objects as well. To use and define `ostringstream` objects the header file `sstream` must be included. In addition the class `ostringstream` offers the following constructors and members:

- `ostringstream ostr(string const &init, ios::openmode mode = ios::out)`:

  when specifying `openmode` as `ios::ate`, the `ostringstream` object is initialized by the `string init` and remaining insertions are appended to the contents of the `ostringstream` object.

- `ostringstream ostr(ios::openmode mode = ios::out)` (this constructor can also be used as default constructor. Alternatively it allows, e.g., forced additions at the end of the information stored in the object so far (using `ios::app`). Example:

  ```
  std::ostringstream out;
  ```

)

- `std::string ostringstream::str() const`:

  a copy of the string that is stored inside the `ostringstream` object is returned.

- `void ostringstream::str(std::string const &str)` (the current object is reinitialized with new initial contents.)

The following example illustrates the use of the `ostringstream` class: several values are inserted into the object. Then, the text contained by the `ostringstream` object is stored in a `std::string`, whose length and contents are thereupon printed. Such `ostringstream` objects are most often used for doing 'type to string' conversions, like converting `int` values to text. Formatting flags can be used with `ostringstreams` as well, as they are part of the `ostream` class.

Here is an example showing an `ostringstream` object being used:

```
#include <iostream>
#include <sstream>

using namespace std;

int main()
{
    ostringstream ostr("hello ", ios::ate);

    cout << ostr.str() << '\n';

    ostr.setf(ios::showbase);
    ostr.setf(ios::hex, ios::basefield);
    ostr << 12345;

    cout << ostr.str() << '\n';

    ostr << " -- ";
    ostr.unsetf(ios::hex);
    ostr << 12;

    cout << ostr.str() << '\n';

    ostr.str("new text");
    cout << ostr.str() << '\n';

    ostr.seekp(4, ios::beg);
    ostr << "world";
    cout << ostr.str() << '\n';
}
/*
    Output from this program:
hello
hello 0x3039
hello 0x3039 -- 12
new text
new world
*/
```

6.5 Input

In **C++** input is primarily based on the `std::istream` class. The `istream` class defines the basic operators and members extracting information from streams: the *extraction operator* (`>>`), and special members like `istream::read` reading unformatted information from streams.

The class `istream` acts as *base class* for several other classes, all offering the functionality of the `istream` class, but adding their own specialties. In the upcoming sections we will introduce:

- The class `istream`, offering the basic facilities for doing input;

- The class `ifstream`, allowing us to read files (comparable to **C**'s `fopen(filename, "r")`);
- The class `istringstream`, allowing us to read information from text that is not stored on files (streams) but in memory (comparable to **C**'s `sscanf` function).

6.5.1 Basic input: the class 'istream'

The class `istream` defines basic input facilities. The `cin` object, is an `istream` object. All facilities related to input as defined by the `ios` class are also available in the `istream` class.

We may define `istream` objects using the following *istream constructor*:

- `istream object(streambuf *sb)`:

  this constructor can be used to construct a wrapper around an existing `std::streambuf` object. Similarly to `ostream` objects, `istream` objects may be defined by passing it initially a 0-pointer. See section 6.4.1 for a discussion, and chapter 23 for examples.

To define the `istream` class in **C++** sources, the `istream` header file must be included. To use the predefined `istream` object `cin`, the `iostream` header file must be included.

6.5.1.1 Reading from 'istream' objects

The class `istream` supports both formatted and unformatted *binary input*. The *extraction operator* (`operator»()`) is used to extract values in a type safe way from `istream` objects. This is called formatted input, whereby human-readable ASCII characters are converted, according to certain formatting rules, to binary values.

The extraction operator points to the objects or variables which receive new values. The normal associativity of >> remains unaltered, so when a statement like

```
cin >> x >> y;
```

is encountered, the leftmost two operands are evaluated first (`cin` >> `x`), and an `istream &` object, which is actually the same `cin` object, is returned. Now, the statement is reduced to

```
cin >> y
```

and the `y` variable is extracted from `cin`.

The >> operator has many (overloaded) variants and thus many types of variables can be extracted from `istream` objects. There is an overloaded >> available for the extraction of an `int`, of a `double`, of a string, of an array of characters, possibly to a pointer, etc. etc.. String or character array extraction by default first skips all white space characters, and will then extract all consecutive non-white space characters. Once an extraction operator has been processed the `istream` object from which the information was extracted is returned and it can immediately be used for additional `istream` operations that appear in the same expression.

Streams lack facilities for formatted input (as used by, e.g., **C**'s `scanf` and `vscanf` functions). Although it is not difficult to add these facilities to the world of streams, `scanf`-like functionality is hardly ever required in **C++** programs. Furthermore, as it is potentially type-*unsafe*, it might be better to avoid formatted input completely.

When binary files must be read, the information should normally not be formatted: an `int` value should be read as a series of unaltered bytes, not as a series of ASCII numeric characters 0 to 9. The following member functions for reading information from `istream` objects are available:

- `int istream::gcount() const`:

  the number of characters read from the input stream by the last unformatted input operation is returned.

- `int istream::get()`:

  the next available single character is returned as an unsigned `char` value using an `int` return type. `EOF` is returned if no more character are available.

- `istream &istream::get(char &ch)`:

  the next single character read from the input stream is stored in `ch`. The member function returns the stream itself which may be inspected to determine whether a character was obtained or not.

- `istream& istream::get(char *buffer, int len, char delim = '\n')`:

  At most `len - 1` characters are read from the input stream into the array starting at `buffer`, which should be at least `len` bytes long. Reading also stops when the delimiter `delim` is encountered. However, the delimiter itself is *not removed* from the input stream.

  Having atored the characters into `buffer`, an `ASCII-Z` character is written beyond the last character stored into the `buffer`. The functions `eof` and `fail` (see section 6.3.1) return 0 (`false`) if the delimiter was encountered before reading `len - 1` characters or if the delimiter was not encountered after reading `len - 1` characters. It is OK to specifiy an `ASCII-Z` delimiter: this way strings terminating in `ASCII-Z` characters may be read from a (binary) file.

- `istream& istream::getline(char *buffer, int len, char delim = '\n')`:

  this member function operates analogously to the `get` member function, but `getline` removes `delim` from the stream if it is actually encountered. The delimiter itself, if encountered, is *not* stored in the `buffer`. If `delim` was *not* found (before reading `len - 1` characters) the `fail` member function, and possibly also `eof` returns true. Realize that the `std::string` class also offers a function `std::getline` which is generally preferred over this `getline` member function that is described here (see section 5.2.4).

- `istream& istream::ignore()`:

  one character is skipped from the input stream.

- `istream& istream::ignore(int n)`:

  `n` characters are skipped from the input stream.

- `istream& istream::ignore(int n, int delim)`:

  at most `n` characters are skipped but skipping characters stops after having removed `delim` from the input stream.

- `int istream::peek()`:

  this function returns the next available input character, but does not actually remove the character from the input stream. `EOF` is returned if no more characters are available.

- `istream& istream::putback(char ch)`:

  The character `ch` is 'pushed back' into the input stream, to be read again as the next available character. `EOF` is returned if this is not allowed. Normally, it is OK to put back one character. Example:

```
string value;
cin >> value;
cin.putback('X');
                // displays: X
cout << static_cast<char>(cin.get());
```

- `istream &istream::read(char *buffer, int len)`:

  At most `len` bytes are read from the input stream into the buffer. If `EOF` is encountered first, fewer bytes are read, with the member function `eof` returning `true`. This function is commonly used when reading *binary* files. Section 6.5.2 contains an example in which this member function is used. The member function `gcount()` may be used to determine the number of characters that were retrieved by `read`.

- `istream& istream::readsome(char *buffer, int len)`:

  at most `len` bytes are read from the input stream into the buffer. All available characters are read into the buffer, but if `EOF` is encountered, fewer bytes are read, without setting the `ios::eofbit` or `ios::failbit`.

- `istream &istream::unget()`:

  the last character that was read from the stream is put back.

6.5.1.2 'istream' positioning

Although not every `istream` object supports repositioning, some do. This means that it is possible to read the same section of a stream repeatedly. Repositioning is frequently used in *database applications* where it must be possible to access the information in the database randomly.

The current position can be obtained and modified using the following members:

- `ios::i(pos_type) istream::tellg()`:

  the current (absolute) position in the file where the next read-operation to the stream will take place is returned.

- `istream &istream::seekg(ios::off_type step, ios::seekdir org)`:

  modifies a stream's actual position. The function expects an `off_type step` representing the number of bytes the current stream position is moved with respect to `org`. The `step` value may be negative, zero or positive.

  The origin of the step, `org` is a value in the `ios::seekdir` enumeration.

  The origin of the step, `org` is a value in the `ios::seekdir` enumeration. Its values are:

  - `ios::beg`:
    the stepsize is computed relative to the beginning of the stream. This value is used by default.
  - `ios::cur`:
    the stepsize is computed relative to the current position of the stream (as returned by `tellp`).

  itttq(end)(ios::end) (the stepsize is interpreted relative to the current end position of the the stream.)

  It is OK to seek beyond the last file position. Seeking before `ios::beg` raises the `ios::fail` flag.

6.5.2 Input from files: the class 'ifstream'

The `std::ifstream` class is derived from the `istream` class: it has the same capabilities as the `istream` class, but can be used to access files for reading.

In order to use the `ifstream` class in **C++** sources, the `fstream` header file must be included. Including `fstream` will not automatically make available the standard streams `cin`, `cout` and `cerr`. Include `iostream` to declare these standard streams.

The following constructors are available for `ifstream` objects:

- `ifstream object`:

  this is the basic constructor. It defines an `ifstream` object which may be associated with an actual file later, using its `open()` member (see below).

- `ifstream object(char const *name, ios::openmode mode = ios::in)`:

  this constructor can be used to define an `ifstream` object and associate it immediately with the file named `name` using output mode `mode`. Section 6.4.2.1 provides an overview of available output modes. Example:

  ```
  ifstream out("/tmp/scratch");
  ```

Instead of directly associating an `ifstream` object with a file, the object can be constructed first, and opened later.

- `void ifstream::open(char const *name, ios::openmode mode = ios::in)`:

  this member function is used to associate an `ifstream` object with an actual file. If the `ios::fail` flag was set before calling `open` and opening succeeds the flag is cleared. Opening an already open stream fails. To reassociate a stream with another file it must first be closed:

  ```
  ifstream in("/tmp/in");
  in >> variable;
  in.close();        // closes in
  in.open("/tmp/in2");
  in >> anotherVariable;
  ```

- `void ifstream::close()`:

  an `ifstream` object is closed by this member function. The function sets the `ios::fail` flag of the closed object. Closing the file will flush any buffered information to the associated file. A file is automatically closed when the associated `ifstream` object ceases to exist.

- `bool ifstream::is_open() const`:

  assume a stream was properly constructed, but it has not yet been attached to a file. E.g., the statement `ifstream ostr` was executed. When we now check its status through `good()`, a non-zero (i.e., *OK*) value is returned. The 'good' status here indicates that the stream object has been constructed properly. It doesn't mean the file is also open. To test whether a stream is actually open, `is_open` should be called. If it returns `true`, the stream is open. Also see the example in section 6.4.2. The following example illustrates reading from a binary file (see also section 6.5.1.1):

  ```
  #include <fstream>
  using namespace std;

  int main(int argc, char **argv)
  {
      ifstream in(argv[1]);
      double   value;

      // reads double in raw, binary form from file.
      in.read(reinterpret_cast<char *>(&value), sizeof(double));
  }
  ```

6.5.3 Input from memory: the class 'istringstream'

To read information from memory using `stream` facilities, `std::istringstream` objects should be used. As the class `istringstream` is derived from the class `ostream` all `ostream`'s facilities are available to `istringstream` objects as well. To use and define `istringstream` objects the header file `sstream` must be included. In addition the class `istringstream` offers the following constructors and members:

- `istringstream istr(string const &init, ios::openmode mode = ios::in)`:

  the object is initialized with `init`'s contents

- `istringstream istr(ios::openmode mode = ios::in)` (this constructor can is usually used as the default constructor. Example:

  ```
  std::istringstream in;
  ```

)

- `void ostringstream::str(std::string const &str)`:

  the current object is reinitialized with new initial contents.

The following example illustrates the use of the `istringstream` class: several values are extracted from the object. Such `istringstream` objects are most often used for doing 'string to type' conversions, like converting text to `int` values (cf. **C**'s `atoi` function). Formatting flags can be used with `ostringstreams` as well, as they are part of the `istream` class. In the example note especially the use of the member `seekg`:

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream istr("123 345");  // store some text.
    int x;

    istr.seekg(2);              // skip "12"
    istr >> x;                  // extract int
    cout << x << '\n';          // write it out
    istr.seekg(0);              // retry from the beginning
    istr >> x;                  // extract int
    cout << x << '\n';          // write it out
    istr.str("666");            // store another text
    istr >> x;                  // extract it
    cout << x << '\n';          // write it out
}
/*
    output of this program:
3
123
666
*/
```

6.5.4 Copying streams

Usually, files are copied either by reading a source file character by character or line by line. The basic *mold* to process streams is as follows:

- Continuous loop:

    1. read from the stream
    2. if reading did not succeed (i.e., `fail` returns `true`), `break` from the loop
    3. process the information that was read

Note that reading must *precede* testing, as it is only possible to know after actually attempting to read from a file whether the reading succeeded or not. Of course, variations are possible: `getline(istream &, string &)` (see section 6.5.1.1) returns an `istream &`, so here reading and testing may be contracted using one expression. Nevertheless, the above mold represents the general case. So, the following program may be used to copy `cin` to `cout`:

```
#include <iostream>
using namespace::std;

int main()
{
    while (true)
    {
        char c;

        cin.get(c);
        if (cin.fail())
            break;
        cout << c;
    }
}
```

Contraction is possible here by combining `get` with the `if`-statement, resulting in:

```
    if (!cin.get(c))
        break;
```

Even so, this would still follow the basic rule: 'read first, test later'.

Simply copying a file isn't required very often. More often a situation is encountered where a file is processed up to a certain point, followed by plain copying the file's remaining information. The next program illustrates this. Using `ignore` to skip the first line (for the sake of the example it is assumed that the first line is at most 80 characters long), the second statement uses yet another overloaded version of the <<-operator, in which a `streambuf` pointer is inserted into a stream. As the member `rdbuf` returns a stream's `streambuf *`, we have a simple means of inserting a stream's contents into an `ostream`:

```
    #include <iostream>
    using namespace std;

    int main()
    {
        cin.ignore(80, '\n');   // skip the first line and...
        cout << cin.rdbuf();    // copy the rest through the streambuf *
    }
```

This way of copying streams only assumes the existence of a `streambuf` object. Consequently it can be used with all specializations of the `streambuf` class.

6.5.5 Coupling streams

Ostream objects can be *coupled* to `ios` objects using the `tie` member function. Tying results in flushing the `ostream`'s buffer whenever an input or output operation is performed on the `ios` object to which the `ostream` object is tied. By default `cout` is tied to `cin` (using `cin.tie(cout)`). This tie means that whenever an operation on `cin` is requested, `cout` is flushed first. To break the tie, `ios::tie(0)` can be called. In the example: `cin.tie(0)`.

Another useful coupling of streams is shown by the tie between `cerr` and `cout`. Because of the tie standard output and error messages written to the screen are shown in sync with the time at which they were generated:

```
#include <iostream>
using namespace std;

int main()
{
    cerr.tie(0);        // untie
    cout << "first (buffered) line to cout ";
    cerr << "first (unbuffered) line to cerr\n";
    cout << "\n";

    cerr.tie(&cout);    // tie cout to cerr
    cout << "second (buffered) line to cout ";
    cerr << "second (unbuffered) line to cerr\n";
    cout << "\n";
}
/*
    Generated output:

    first (unbuffered) line to cerr
    first (buffered) line to cout
    second (buffered) line to cout second (unbuffered) line to cerr
*/
```

An alternative way to couple streams is to make streams use a common `streambuf` object. This can be implemented using the `ios::rdbuf(streambuf *)` member function. This way two streams can use, e.g. their own formatting, one stream can be used for input, the other for output, and redirection using the stream library rather than operating system calls can be implemented. See the next sections for examples.

6.6 Advanced topics

6.6.1 Redirecting streams

Using `ios::rdbuf` streams can be forced to share their `streambuf` objects. Thus information written to one stream is actually written to another stream; a phenomenon normally called *redirection*. Redirection is commonly implemented at the operating system level, and sometimes that is still necessary (see section 23.4.1).

A common situation where redirection is useful is when error messages should be written to file rather than to the standard error stream, usually indicated by its file descriptor number 2. In the Unix operating system using the `bash` shell, this can be realized as follows:

```
program 2>/tmp/error.log
```

Following this command any error messages written by `program` are saved on the file `/tmp/error.log`, instead of appearing on the screen.

Here is an example showing how this can be implemented using `streambuf` objects. Assume `program` expects an argument defining the name of the file to write the error messages to. It could be called as follows:

```
program /tmp/error.log
```

The program looks like this, an explanation is provided below the program's source text:

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char **argv)
{
    ofstream errlog;                                    // 1
    streambuf *cerr_buffer = 0;                         // 2

    if (argc == 2)
    {
        errlog.open(argv[1]);                           // 3
        cerr_buffer = cerr.rdbuf(errlog.rdbuf());       // 4
    }
    else
    {
        cerr << "Missing log filename\n";
        return 1;
    }

    cerr << "Several messages to stderr, msg 1\n";
    cerr << "Several messages to stderr, msg 2\n";

    cout << "Now inspect the contents of " <<
            argv[1] << "... [Enter] ";
    cin.get();                                          // 5

    cerr << "Several messages to stderr, msg 3\n";

    cerr.rdbuf(cerr_buffer);                            // 6
    cerr << "Done\n";                                   // 7
}
/*
    Generated output on file argv[1]

    at cin.get():

Several messages to stderr, msg 1
Several messages to stderr, msg 2

    at the end of the program:

Several messages to stderr, msg 1
Several messages to stderr, msg 2
Several messages to stderr, msg 3
*/
```

- At lines 1-2 local variables are defined: `errlog` is the `ofstream` to write the error messages too,

and `cerr_buffer` is a pointer to a `streambuf`, to point to the original `cerr` buffer.

- At line 3 the alternate error stream is opened.
- At line 4 redirection takes place: `cerr` will write to the `streambuf` defined by `errlog`. It is important that the original buffer used by `cerr` is saved, as explained below.
- At line 5 we pause. At this point, two lines were written to the alternate error file. We get a chance to take a look at its contents: there were indeed two lines written to the file.
- At line 6 the redirection is terminated. This is very important, as the `errlog` object is destroyed at the end of `main`. If `cerr`'s buffer would not have been restored, then at that point `cerr` would refer to a non-existing `streambuf` object, which might produce unexpected results. It is the responsibility of the programmer to make sure that an original `streambuf` is saved before redirection, and is restored when the redirection ends.
- Finally, at line 7, `Done` is again written to the screen, as the redirection has been terminated.

6.6.2 Reading AND Writing streams

In order to both read and write to a stream an `std::fstream` object must be created. As with `ifstream` and `ofstream` objects, its constructor receives the name of the file to be opened:

```
fstream inout("iofile", ios::in | ios::out);
```

Note the use of the constants `ios::in` and `ios::out`, indicating that the file must be opened for both reading and writing. Multiple mode indicators may be used, concatenated by the `bitor` operator. Alternatively, instead of `ios::out`, `ios::app` could have been used and mere writing would become appending (at the end of the file).

Reading and writing to the same file is always a bit awkward: what to do when the file may not yet exist, but if it already exists it should not be rewritten? Having fought with this problem for some time I now use the following approach:

```
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    fstream rw("fname", ios::out | ios::in);

    if (!rw)            // file didn't exist yet
    {
        rw.clear();     // try again, creating it using ios::trunc
        rw.open("fname", ios::out | ios::trunc | ios::in);
    }

    if (!rw)            // can't even create it: bail out
    {
        cerr << "Opening 'fname' failed miserably" << '\n';
        return 1;
    }

    cerr << "We're at: " << rw.tellp() << '\n';

                        // write something
```

```
        rw << "Hello world" << '\n';

        rw.seekg(0);        // go back and read what's written

        string s;
        getline(rw, s);

        cout << "Read: " << s << '\n';
    }
```

Under this approach if the first construction attempt fails `fname` doesn't exist yet. But then `open` can be attempted using the `ios::trunc` flag. If the file already existed, the construction would have succeeded. By specifying `ios::ate` when defining `rw`, the initial read/write action would by default have taken place at `EOF`.

Under **DOS**-like operating systems that use the multiple character sequence `\r\n` to separate lines in text files the flag `ios::binary` is required to process binary files ensuring that `\r\n` combinations are processed as two characters. In general, `ios::binary` should be specified when binary (non-text) files are to be processed. By default files are opened as text files. Unix operating systems do not distinguish text files from binary files.

With `fstream` objects, combinations of file flags are used to make sure that a stream is or is not (re)created empty when opened. See section 6.4.2.1 for details.

Once a file has been opened in read and write mode, the << operator can be used to insert information into the file, while the >> operator may be used to extract information from the file. These operations may be performed in any order, but a `seekg()` or `seekp()` operation is required when switching between insertions and extractions. The example shows a white space delimited word being read from a file, writing another string to the file, just beyond the point where the just read word terminated. Finally yet another string is read which is found just beyond the location where the just written strings ended:

```
fstream f("filename", ios::in | ios::out);
string  str;

f >> str;       // read the first word

                // write a well known text
f.seekg(0, ios::cur);
f << "hello world";

f.seekp(0, ios::cur);
f >> str;       // and read again
```

Since a *seek* or *clear* operation is required when alternating between read and write (extraction and insertion) operations on the same file it is not possible to execute a series of << and >> operations in one expression statement.

Of course, random insertions and extractions are hardly ever used. Generally, insertions and extractions occur at well-known locations in a file. In those cases, the position where insertions or extractions are required can be controlled and monitored by the `seekg`, `seekp`, `tellg` and `tellp` members (see sections 6.4.1.2 and 6.5.1.2).

Error conditions (see section 6.3.1) occurring due to, e.g., reading beyond end of file, reaching end of file, or positioning before begin of file, can be cleared by the `clear` member function. Following `clear` processing may continue. E.g.,

```
fstream f("filename", ios::in | ios::out);
string  str;
```

```
    f.seekg(-10);   // this fails, but...
    f.clear();      // processing f continues

    f >> str;       // read the first word
```

A situation where files are both read and written is seen in *database* applications, using files consisting of records having fixed sizes, and where locations and sizes of pieces of information are known. For example, the following program adds text lines to a (possibly existing) file. It can also be used to retrieve a particular line, given its order-number in the file. A *binary file* `index` allows for the quick retrieval of the location of lines.

```
    #include <iostream>
    #include <fstream>
    #include <string>
    using namespace std;

    void err(char const *msg)
    {
        cout << msg << '\n';
    }

    void err(char const *msg, long value)
    {
        cout << msg << value << '\n';
    }

    void read(fstream &index, fstream &strings)
    {
        int idx;

        if (!(cin >> idx))                          // read index
            return err("line number expected");

        index.seekg(idx * sizeof(long));            // go to index-offset

        long offset;

        if
        (
            !index.read                             // read the line-offset
            (
                reinterpret_cast<char *>(&offset),
                sizeof(long)
            )
        )
            return err("no offset for line", idx);

        if (!strings.seekg(offset))                 // go to the line's offset
            return err("can't get string offset ", offset);

        string line;

        if (!getline(strings, line))                // read the line
            return err("no line at ", offset);

        cout << "Got line: " << line << '\n';       // show the line
    }

    void write(fstream &index, fstream &strings)
```

```
{
    string line;

    if (!getline(cin, line))                // read the line
        return err("line missing");

    strings.seekp(0, ios::end);             // to strings
    index.seekp(0, ios::end);               // to index

    long offset = strings.tellp();

    if
    (
        !index.write                        // write the offset to index
        (
            reinterpret_cast<char *>(&offset),
            sizeof(long)
        )
    )
        return err("Writing failed to index: ", offset);

    if (!(strings << line << '\n'))         // write the line itself
        return err("Writing to 'strings' failed");
                                            // confirm writing the line
    cout << "Write at offset " << offset << " line: " << line << '\n';
}

int main()
{
    fstream index("index", ios::trunc | ios::in | ios::out);
    fstream strings("strings", ios::trunc | ios::in | ios::out);

    cout << "enter 'r <number>' to read line <number> or "
                              "w <line>' to write a line\n"
            "or enter 'q' to quit.\n";

    while (true)
    {
        cout << "r <nr>, w <line>, q ? ";       // show prompt

        index.clear();
        strings.clear();

        string cmd;
        cin >> cmd;                             // read cmd

        if (cmd == "q")                         // process the cmd.
            return 0;

        if (cmd == "r")
            read(index, strings);
        else if (cmd == "w")
            write(index, strings);
        else
            cout << "Unknown command: " << cmd << '\n';
    }
}
```

Another example showing reading *and* writing of files is provided by the next program. It also illustrates the processing of ASCII-Z delimited strings:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{                                       // r/w the file
    fstream f("hello", ios::in | ios::out | ios::trunc);

    f.write("hello", 6);                // write 2 ascii-z strings
    f.write("hello", 6);

    f.seekg(0, ios::beg);               // reset to begin of file

    char buffer[100];                   // or: char *buffer = new char[100]
    char c;
                                        // read the first 'hello'
    cout << f.get(buffer, sizeof(buffer), 0).tellg() << endl;;
    f >> c;                             // read the ascii-z delim

                                        // and read the second 'hello'
    cout << f.get(buffer + 6, sizeof(buffer) - 6, 0).tellg() << endl;

    buffer[5] = ' ';                    // change asciiz to ' '
    cout << buffer << endl;             // show 2 times 'hello'
}
/*
    Generated output:
5
11
hello hello
*/
```

A completely different way to read and write streams may be implemented using `streambuf` members. All considerations mentioned so far remain valid (e.g., before a read operation following a write operation `seekg` must be used). When `streambuf` objects are used, either an `istream` is associated with the `streambuf` object of another `ostream` object, or an `ostream` object is associated with the `streambuf` object of another `istream` object. Here is the previous program again, now using *associated streams*:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

void err(char const *msg);      // see earlier example
void err(char const *msg, long value);

void read(istream &index, istream &strings)
{
    index.clear();
    strings.clear();

    // insert the body of the read() function of the earlier example
}
```

```
void write(ostream &index, ostream &strings)
{
    index.clear();
    strings.clear();

    // insert the body of the write() function of the earlier example
}

int main()
{
    ifstream index_in("index", ios::trunc | ios::in | ios::out);
    ifstream strings_in("strings", ios::trunc | ios::in | ios::out);
    ostream  index_out(index_in.rdbuf());
    ostream  strings_out(strings_in.rdbuf());

    cout << "enter 'r <number>' to read line <number> or "
                              "w <line>' to write a line\n"
            "or enter 'q' to quit.\n";

    while (true)
    {
        cout << "r <nr>, w <line>, q ? ";       // show prompt

        string cmd;

        cin >> cmd;                             // read cmd

        if (cmd == "q")                         // process the cmd.
            return 0;

        if (cmd == "r")
            read(index_in, strings_in);
        else if (cmd == "w")
            write(index_out, strings_out);
        else
            cout << "Unknown command: " << cmd << endl;
    }
}
```

In this example

- the streams associated with the `streambuf` objects of existing streams are not `ifstream` or `ofstream` objects but basic `istream` and `ostream` objects.
- The `streambuf` object is not defined by an `ifstream` or `ofstream` object. Instead it is defined outside of the streams, using a `filebuf` (cf. section 14.7.2) and constructions like:

```
filebuf fb("index", ios::in | ios::out | ios::trunc);
istream index_in(&fb);
ostream index_out(&fb);
```

- An `ifstream` object can be constructed using stream modes normally used with `ofstream` objects. Conversely, an `ofstream` objects can be constructed using stream modes normally used with `ifstream` objects.
- If `istream` and `ostreams` share a `streambuf`, then their read and write pointers (should) point to the shared buffer: they are tightly coupled.
- The advantage of using an external (separate) `streambuf` over a predefined `fstream` object is (of course) that it opens the possibility of using `stream` objects with specialized `streambuf` ob-

jects. These `streambuf` objects may specifically be constructed to control and interface particular devices. Elaborating this (see also section 14.7) is left as an exercise to the reader.

Chapter 7

Classes

In this chapter classes are introduced. Two special member functions, the constructor and the destructor, are presented.

In steps we will construct a class `Person`, which could be used in a database application to store a person's name, address and phone number.

Let's start by creating a `class Person` right away. From the onset, it is important to make the distinction between the class *interface* and its *implementation*. A class may loosely be defined as 'a set of data and all the functions operating on those data'. This definition will be refined later but for now it is sufficient to get us started.

A class interface is a definition, defining the organization of objects of that class. Normally a definition results in memory reservation. E.g., when defining `int variable` the compiler will make sure that some memory will be reserved in the final program storing `variable`'s values. Although it is a definition no memory is set aside by the compiler once it has processed the class definition. But a class definition follows the *one definition rule*: in **C++** entities may be defined only once. As a *class definition* does not imply that memory is being reserved the term *class interface* is preferred instead.

Class interfaces are normally contained in a class *header file*, e.g., `person.h`. We'll start our `class Person` interface here:

```
#include <string>

class Person
{
    std::string d_name;         // name of person
    std::string d_address;      // address field
    std::string d_phone;        // telephone number
    size_t      d_weight;       // the weight in kg.

    public:                     // member functions
        void setName(std::string const &name);
        void setAddress(std::string const &address);
        void setPhone(std::string const &phone);
        void setWeight(size_t weight);

        std::string const &name()    const;
        std::string const &address() const;
        std::string const &phone()   const;
        size_t weight()              const;
};
```

The member functions that are *declared* in the interface must still be implemented. The implementa-

tion of these members is properly called their definition.

In addition to the member function a class defines the data manipulated by the member functions. These data are called the *data members*. In `Person` they are `d_name, d_address, d_phone` and `d_weight`. Data members should be given private access rights. Since the class uses private access rights by default they may simply be listed at the top of the interface.

All communication between the outer world and the class data is routed through the class's member functions. Data members may receive new values (e.g., using `setName`) or they may be retrieved for inspection (e.g., using `name`). Functions merely returning values stored inside the object, not allowing the caller to modify these internally stored values, are called *accessors*.

Syntactically there is only a marginal difference between a class and a struct. Classes by default define *private* members, structs define *public* members. Conceptually, though, there are difference. In **C++** structs are used in the way they are used in **C**: to aggregate data, which are all freely accessible. Classes, on the other hand, hide their data from access by the outside world (which is aptly called *data hiding*) and offer member functions to define the communication between the outer world and the class's data members.

Following *Lakos* (Lakos, J., 2001) **Large-Scale C++ Software Design** (Addison-Wesley) I suggest the following setup of class interfaces:

- All data members have *private access rights*, and are placed at the top of the interface.
- All data members start with `d_`, followed by a name suggesting their meaning (in chapter 11 we'll also encounter data members starting with `s_`).
- Non-private data members *do* exist, but one should be hesitant to define non-private access rights for data members (see also chapter 13).
- Two broad categories of member functions are *manipulators* and *accessors*. *Manipulators* allow the users of objects to modify the internal data of the objects. By convention, manipulators start with `set`. E.g., `setName`.
- With *accessors*, a `get`-prefix is still frequently encountered, e.g., `getName`. However, following the conventions promoted by the **Qt** (see `http://www.trolltech.com`) *Graphical User Interface Toolkit*, the `get`-prefix is now deprecated. So, rather than defining the member `getAddress`, it should simply be named `address`.
- Normally (exceptions exist) the public member functions of a class are listed first, immediately following the class's data members, They are the important elements of the interface as they define the features the class is offering to its users. It's a matter of convention to list them high up in the interface. The keyword `private` is needed beyond the public members to switch back from public members to private access rights which nicely separates the members that may be used 'by the general public' from the class's own support members.

Style conventions usually take a long time to develop. There is nothing obligatory about them, though. I suggest that readers who have compelling reasons *not* to follow the above style conventions use their own. All others are strongly advised to adopt the above style conventions.

7.1 The constructor

C++ classes may contain two special categories of member functions which are essential to the proper working of the class. These categories are the constructors and the destructor. The *destructor*'s primary task is to return memory allocated by an object to the common pool when an object goes 'out of scope'. Allocation of memory is discussed in chapter 8, and destructors will therefore be discussed in depth in that chapter. In this chapter the emphasis will be on the class's organization and its constructors.

Constructor are recognized by their names which is equal to the class name. Constructors do not specify return values, not even `void`. E.g., the class `Person` may define a constructor `Person::Person()`.

The **C++** run-time system ensures that the constructor of a class is called when a variable of the class is defined. It is possible to define a class lacking any constructor. In that case the compiler will define a default constructor that is called when an object of that class is defined. What actually happens in that case depends on the data members that are defined by that class (cf. section 7.2.1).

Objects may be defined locally or globally. However, in **C++** most objects are defined locally. Globally defined objects are hardly ever required and are somewhat deprecated.

When a local object is defined its constructor is called every time the function is called. The object's constructor is activated at the point where the object is defined (a subtlety is that an object may be defined implicitly as, e.g., a temporary variable in an expression).

When an object is defined as a static object is called when the program starts. In this case the constructor is called even before the function `main` starts. Example:

```
#include <iostream>
using namespace std;

class Demo
{
    public:
        Demo();
};

Demo::Demo()
{
    cout << "Demo constructor called\n";
}

Demo d;

int main()
{}

/*
    Generated output:
Demo constructor called
*/
```

The program contains one global object of the class `Demo` with `main` having an empty body. Nonetheless, the program produces some output generated by the constructor of the globally defined `Demo` object.

Constructors have a very important and well-defined role. They must ensure that all the class's data members have sensible or at least well-defined values once the object has been constructed. We'll get back to this important task shortly. The *default constructor* has no argument. It is defined by the compiler unless another constructor is defined and unless its definition is suppressed (cf. section 7.4). If a default constructor is required in addition to another constructor then the default constructor must explicitly be defined as well. The C++0x standard provides special syntax to do that as well, which is also covered by section 7.4.

7.1.1 A first application

Our example class `Person` has three string data members and a `size_t d_weight` data member. Access to these data members is controlled by interface functions.

Whenever an object is defined the class's constructor(s) ensure that its data members are given 'sensible' values. Thus, objects never suffer from uninitialized values. Datamembers may be given new values, but that should never be directly allowed. It is a core principle (called *data hiding*) of good class design dat its data members are private. The modification of data members is therefore fully

controlled by member functions and thus, indirectly, by the class-designer. The class *encapsulates* all actions performed on its data members and due to this *encapsulation* the class object may assume the 'responsibility' for its own data-integrity. Here is a minimal definition of `Person`'s manipulating members:

```
#include "person.h"                 // given earlier

void Person::setName(string const &name)
{
    d_name = name;
}
void Person::setAddress(string const &address)
{
    d_address = address;
}
void Person::setPhone(string const &phone)
{
    d_phone = phone;
}
void Person::setWeight(size_t weight)
{
    d_weight = weight;
}
```

It's a minimal definition in that no checks are performed. But it should be clear that checks are easy to implement. E.g., to ensure that a phone number only contains digits one could define:

```
void Person::setPhone(string const &phone)
{
    if (phone.find_first_not_of("0123456789") == string::npos)
        d_phone = phone;
    else
        cout << "A phone number may only contain digits\n";
}
```

Similarly, access to the data members is controlled by encapsulating *accessor* members. Accessors ensure that data members cannot suffer from uncontrolled modifications. Since accessors conceptually do not modify the object's data (but only retrieve the data) these member functions are given the predicate `const`. They are called *const member functions*, which, as they are guaranteed not to modify their object's data, are available to both modifiable and constant objects (cf. section 7.5).

To prevent *backdoors* we must also make sure that the data mamber is not modifiable through an accessor's return value. For values of built-in primitive types that's easy, as they are usually returned by value, which are copies of the values found in variables. But since objects may be fairly large making copies are usually prevented by returning objects by reference. A backdoor is created by returning a data member by reference, as in the following example, showing the allowed abuse below the function definition:

```
string &Person::name() const
{
    return d_name;
}

Person somebody;
somebody.setName("Nemo");

somebody.name() = "Eve";    // Oops, backdoor changing the name
```

To prevent the backdoor objects are returned as *const references* from accessors. Here are the implementations of Person's accessors:

```
    #include "person.h"                // given earlier

    string const &Person::name() const
    {
        return d_name;
    }
    string const &Person::address() const
    {
       return d_address;
    }
    string const &Person::phone() const
    {
       return d_phone;
    }
    size_t Person::weight() const
    {
       return d_weight;
    }
```

The Person class interface remains the starting point for the class design: its memberfunctions define what can be asked of a Person object. In the end the implementation of its members merely is a technicality allowing Person objects to do their jobs.

The next example shows how the class Person may be used. An object is initialized and passed to a function printperson(), printing the person's data. Note the reference operator in the parameter list of the function printperson. Only a reference to an existing Person object is passed to the function, rather than a complete object. The fact that printperson does not modify its argument is evident from the fact that the parameter is declared const.

```
    #include <iostream>
    #include "person.h"                // given earlier

    void printperson(Person const &p)
    {
        cout << "Name    : " << p.name()     << "\n"
                "Address : " << p.address()  << "\n"
                "Phone   : " << p.phone()    << "\n"
                "Weight  : " << p.weight()   << '\n';
    }

    int main()
    {
        Person p;

        p.setName("Linus Torvalds");
        p.setAddress("E-mail: Torvalds@cs.helsinki.fi");
        p.setPhone(" - not sure - ");
        p.setWeight(75);           // kg.

        printperson(p);
    }
/*
    Produced output:

Name    : Linus Torvalds
Address : E-mail: Torvalds@cs.helsinki.fi
```

```
Phone   :  - not sure -
Weight  : 75

*/
```

7.1.2 Constructors: with and without arguments

The class `Person`'s constructor so far has no parameters. **C++** allows constructors to be defined with or without parameter lists. The arguments are supplied when an object is defined.

For the class `Person` a constructor expecting three strings and a `size_t` might be useful. Represening, respectively, the person's name, address, phone number and weight. This constructor is:

```
Person::Person(string const &name, string const &address,
               string const &phone, size_t weight)
{
    d_name = name;
    d_address = address;
    d_phone = phone;
    d_weight = weight;
}
```

It must of course also be declared in the class interface:

```
class Person
{
    // data members (not altered)

    public:
        Person(std::string const &name, std::string const &address,
               std::string const &phone, size_t weight);

        // rest of the class interface (not altered)
};
```

Now that this constructor has been declared, the default constructor must explicitly be declared as well if we still want to be able to construct a plain `Person` object without any specific initial values for its data members. The class `Person` would thus support two constructors, and the part declaring the constructors now becomes:

```
class Person
{
    // data members
    public:
        Person();
        Person(std::string const &name, std::string const &address,
               std::string const &phone, size_t weight);

        // additional members
};
```

In this case, the default constructor doesn't have to do very much, as it doesn't have to initialize the `string` data members of the `Person` object. As these data members are objects themselves, they are initialized to empty strings by their own default constructor. However, there is also a `size_t` data member. That member is a variable of a built-in type and such variabes do not have constructors

and so are not initialized automatically. Therefore, unless the value of the `d_weight` data member is explicitly initialized it will be

- a *random* value for local `Person` objects;
- 0 for global and static `Person` objects.

The 0-value might not be too bad, but normally we don't want a *random* value for our data members. So, even the default constructor has a job to do: initializing the data members which are not initialized to sensible values automatically. Its implementation can be:

```
Person::Person()
{
    d_weight = 0;
}
```

Using constructors with and without arguments is illustrated next. The object `karel` is initialized by the constructor defining a non-empty parameter list while the default constructor is used with the `anon` object:

```
int main()
{
    Person karel("Karel", "Rietveldlaan 37", "542 6044", 70);
    Person anon;
}
```

The two `Person` objects are defined when `main` starts as they are *local* objects, living only for as long as `main` is active.

If `Person` objects must be definable using other arguments, corresponding constructors must be added to `Person`'s interface. Apart from overloading class constructors it is also possible to provide constructors with default argument values. These default arugments must be specified with the constructor declarations in the class interface, like so:

```
class Person
{
    public:
        Person(std::string const &name,
               std::string const &address = "--unknown--",
               std::string const &phone   = "--unknown--",
               size_t weight = 0);

};
```

Often, constructors use highly similar implementions. This results from the fact that the constructor's parameters are often defined for convenience: a constructor not requiring a `phone` number but requiring a `weight` cannot be defined using default arguments, since `phone` is not the constructor's last parameter. Consequently as special constructor is required lacking `phone` in its parameter list.

In pre C++0x **C++** this situation is commonly tackled as follows: all constructors *must* initialize their reference and `const` data members, or the compiler will (rightfully) complain. To initialize the remaining members (non-const and non-reference members) we have two options:

- If the body of the construction process is sizeable but (parameterizable) identical to other constructors bodies then factorize. Define a private member `init` which is called by the constructors to provide the object's data members with their appropriate values.
- If the constructors act fundamentally differently, then there's nothing to factorize and each constructor must be implemented by itself.

C++0x allows constructors to call each other. This is illustrated in section 7.2.3 below.

7.1.2.1 The order of construction

The possibility to pass arguments to constructors allows us to monitor the construction order of objects during program execution. This is illustrated by the next program using a class `Test`. The program defines a global `Test` object and two local `Test` objects. The order of construction is as expected: first global, then main's first local object, then `func`'s local object, and then, finally, `main`'s second local object:

```
    #include <iostream>
    #include <string>
    using namespace std;

    class Test
    {
        public:
            Test(string const &name);   // constructor with an argument
    };

    Test::Test(string const &name)
    {
        cout << "Test object " << name << " created" << endl;
    }

    Test globaltest("global");

    void func()
    {
        Test functest("func");
    }

    int main()
    {
        Test first("main first");
        func();
        Test second("main second");
    }
/*
    Generated output:
Test object global created
Test object main first created
Test object func created
Test object main second created
*/
```

7.2 Objects inside objects: composition

In the class `Person` objects are used as data members. This construction technique is called *composition*.

Composition is neither extraordinary nor **C++** specific: in **C** a `struct` or `union` field is commonly used in other compound types. In **C++** it requires some special thought as their initialization sometimes is subject to restrictions, as discussed in the next few sections.

7.2.1 Composition and const objects: const member initializers

Unless specified otherwise object data members of classes are initialized by their default constructors. Using the default constructor might not always be the optimal way to intialize an object and it might not even be possible: a class might simply not define a default constructor.

Earlier we've encountered the following constructor of the `Person`:

```
Person::Person(string const &name, string const &address,
               string const &phone, size_t weight)
{
    d_name = name;
    d_address = address;
    d_phone = phone;
    d_weight = weight;
}
```

Think briefly about what is going on in this constructor. In the constructor's body we encounter assignments to string objects. Since assignments are used in the constructor's body their left-hand side objects must exist. But when objects are coming into existence constructors *must* have been called. The initialization of those objects is thereupon immediately undone by the body of `Person`'s constructor. That is not only inefficient but sometimes downright impossible. Assume that the class interface mentions a `string const` data member: a data member whose value is not supposed to change at all (like a birthday, which usually doesn't change very much and is therefore a good candidate for a `string const` data member). Constructing a birthday object and providing it with an initial value is OK, but changing the initial value isn't.

The body of a constructor allows assignments to data members. The *initialization* of data members happens before that. **C++** defines the *member initializer* syntax allowing us to specify the way data members are initialized at construction time. Member initializers are specified as a list of constructor specifications between a colon following a constructor's parameter list and the opening curly brace of a constructor's body, as follows:

```
Person::Person(string const &name, string const &address,
               string const &phone, size_t weight)
:
    d_name(name),
    d_address(address),
    d_phone(phone),
    d_weight(weight)
{}
```

Member initialization *always* occurs when objects are composed in classes: if *no* constructors are mentioned in the member initializer list the default constructors of the objects are called. Note that this only holds true for *objects*. Data members of primitive data types are *not* initialized automatically.

Member initialization can, however, also be used for primitive data members, like `int` and `double`. The above example shows the initialization of the data member `d_weight` from the parameter `weight`. When member initializers are used the data member could even have the same name as the constructor's parameter (although this is deprecated) as there is no ambiguity and the first (left) identifier used in a member initializer is always a data member that is initialized whereas the identifier between parentheses is interpreted as the parameter.

The *order* in which class type data members are initialized is defined by the order in which those members are defined in the composing class interface. If the order of the initialization in the constructor differs from the order in the class interface, the compiler complains, and reorders the initialization so as to match the order of the class interface.

Member initializers should be used as often as possible. As shown it may be required to use them (e.g., to initialize const data members, or to initialize objects of classes lacking default constructors) but *not*

using member initializers also results in inefficient code as the default constructor of a data member is always automatically called unless an explicit member initializer is specified. Reassignment in the constructor's body following default construction is than clearly inefficient. Of course, sometimes it is fine to use the default constructor, but in those cases the explicit member initializer can be omitted.

As a rule of thumb: if a value is assigned to a data member in the constructor's body then try to avoid that assignment in favor of using a member initializer.

7.2.2 Composition and reference objects: reference member initializers

Apart from using member initializers to initialize composed objects (be they `const` objects or not), there is another situation where member initializers must be used. Consider the following situation.

A program uses an object of the class `Configfile`, defined in `main` to access the information in a configuration file. The configuration file contains parameters of the program which may be set by changing the values in the configuration file, rather than by supplying command line arguments.

Assume another object used in `main` is an object of the class `Process`, doing 'all the work'. What possibilities do we have to tell the object of the class `Process` that an object of the class `Configfile` exists?

- The objects could have been declared as *global* objects. This *is* a possibility, but not a very good one, since all the advantages of local objects are lost.
- The `Configfile` object may be passed to the `Process` object at construction time. Bluntly passing an object (i.e., by value) might not be a very good idea, since the object must be copied into the `Configfile` parameter, and then a data member of the `Process` class can be used to make the `Configfile` object accessible throughout the `Process` class. This might involve yet another object-copying task, as in the following situation:

```
Process::Process(Configfile conf)   // a copy from the caller
{
    d_conf = conf;                  // copying to d_conf member
}
```

- The copy-instructions can be avoided if *pointers* to the `Configfile` objects are used, as in:

```
Process::Process(Configfile *conf)  // pointer to external object
{
    d_conf = conf;                  // d_conf is a Configfile *
}
```

  This construction as such is OK, but forces us to use the '`->`' field selector operator, rather than the '`.`' operator, which is (disputably) awkward. Conceptually one tends to think of the `Configfile` object as an object, and not as a pointer to an object. In **C** this would probably have been the preferred method, but in **C++** we can do better.

- Rather than using value or pointer parameters, the `Configfile` parameter could be defined as a *reference parameter* of `Process`'s constructor. Next, use a `Config` reference data member in the `class Process`.

But a reference variable cannot be initialized using an assignment, and so the following is incorrect:

```
Process::Process(Configfile &conf)
{
    d_conf = conf;        // wrong: no assignment
}
```

The statement `d_conf = conf` fails, because it is not an initialization, but an assignment of one `Configfile` object (i.e., `conf`), to another (`d_conf`). An assignment to a reference variable is actually an assignment to the variable the reference variable refers to. But to what variable does `d_conf` refer? To no variable at all, since we haven't initialized `d_conf`. After all, the whole purpose of the statement `d_conf = conf` was to initialize `d_conf`....

How to initialize `d_conf`? We once again use the member initializer syntax. Here is the correct way to initialize `d_conf`:

```
Process::Process(Configfile &conf)
:
    d_conf(conf)      // initializing reference member
{}
```

The above syntax must be used in all cases where reference data members are used. E.g., if `d_ir` would be an `int` reference data member, a construction like

```
Process::Process(int &ir)
:
    d_ir(ir)
{}
```

would have been required.

7.2.3 Constructors calling constructors (C++0x, ?)

Often constructors are specializations of each other, allowing objects to be constructed using subsets of arguments for its data members, and/or using default argument values for other data members. In many cases classes define initialization members that are called by the various constructors. Here is an example. A class `Stat` is designed as a wrapper class around **C**'s **stat**(2) function. It might define three constructors: one expecting no arguments and initializing all data members to appropriate values; a second one doing the same, but it calls `stat` with the filename provided to the constructor and a third one expecting a filename and a search path for the provided file name. Rather than repeating the initialization code in each constructor, the common code can be factorized into a member `init()` which is thereupon called by the constructors.

The C++0x standard offers an alternative to this design by allowing constructors to call each other. The C++0x standard allows constructors to call each other as shown in the following example:

```
class Stat
{
    public:
        Stat()
        :
            // default initialization of members
        {}
        Stat(std::string const &fileName)
        :
            Stat()
        {
            set(fileName);
        }
        Stat(std::string const &fileName, std::string const &searchPath)
        :
            Stat()
        {
            set(fileName, searchPath);
```

```
        }
        ...
};
```

There is one *caveat*: **C++** considers the object as fully constructed once a constructor has normally finished. Once a constructor has finished the class's destructor is guaranteed to be called (cf. chapter 8) and so remaining code must make sure that, e.g., all the class's pointer data members remain in a valid state. Also, as a prelude to chapter 9, realize that a destructor *will* be called if a constructor throws an exception after having completed the call to another constructor.

C++ allows static const integral data members to be initialized within the class interfaces (cf. chapter 11). The C++0x standard adds to this the facility to provide all data members (const or non-const, integral or non-integral) with a default initialization that is specified by the class interface.

These default initializations may be overruled by constructors. E.g., if the class `Stat` uses a data member `bool d_hasPath` which is `false` by default but the third constructor (see above) should initialize it to `true` then the following approach is possible:

```
class Stat
{
    bool d_hasPath = false;

    public:
        Stat(std::string const &fileName, std::string const &searchPath)
        :
            d_hasPath(true)     // overrule the interface-specified
                                // value
        {}
};
```

Here `d_hasPath` receives its value only once: it's always initialized to `false` except when the shown constructor is used in which case it is initialized to `true`.

7.3 Uniform initialization (C++0x, 4.4)

When defining variables and objects are defined they may immediately be given intial values. Class type objects are always initialized using one of their avaiable constructors. **C** already supports the array and struct *initializer list* consisting of a list of constant expressions surrounded by a pair of curly braces. A comparable initialization, called *uniform initialization* is added to **C++** by the C++0x standard. It uses the following syntax:

```
Type object {value list};
```

When defining objects using a list of objects each individual object may use its own uniform initialization.

The advantage of uniform initialization over using constructors is that using constructor arguments may sometimes result in an ambiguity as constructing an object may sometimes be confused with using the object's overloaded function call operator (cf. section 10.10). As initializer lists can only be used with *plain old data* (POD) types (cf. section 8.8) and with classes that are 'initializer list aware' (like `std::vector`) the ambiguity does not arise when initializer lists are used.

Uniform initialization can be used to initialize an object or variable, but also to initialize data members in a constructor or implicitly in the return statement of functions. Examples:

```
class Person
```

```
{
    // data members
    public:
        Person(std::string const &name, size_t weight)
        :
            d_name {name},
            d_weight {weight}
        {}

        Person copy() const
        {
            return {d_name, d_weight};
        }
};
```

Although the uniform intialization syntax is slightly different from the syntax of an initializer list (the latter using the assignment operator) the compiler nevertheless uses the initializer list if a constructor supporting an initializer list is available. As an example consider:

```
class Vector
{
    public:
        Vector(size_t size);
        Vector(std::initializer_list<int> const &values);
};

Vector vi = {4};
```

When defining `vi` the constructor expecting the initializer list is called rather than the constructor expecting a `size_t` argument. If the latter constructor is required the definition using the standard constructor syntax must be used. I.e., `Vector vi(4)`.

Initializer lists are themselves objects that may be constructed using another initializer list. However, values stored in an initializer list are immutable. Once the initializer list has been defined their values remain as-is. Initializer lists support a basic set of member functions and constructors:

- `initializer_list<Type> object`:

  defines `object` as an empty initializer list

- `initializer_list<Type> object { list of Type values }`:

  defines `object` as an initializer list containing `Type` values

- `initializer_list<Type> object(other)`:

  initializes `object` using the values stored in `other`

- `size_t size() const`:

  returns the number of elements in the initializer list

- `Type const *begin() const`:

  returns a pointer to the first element of the initializer list

- `Type const *end() const`:

  returns a pointer just beyond the location of the last element of the initializer list

7.4 Defaulted and deleted class members (C++0x, 4.4)

In everyday class design two situation are frequently encountered:

- A class offering constructors explicitly has to define a default constructor;
- A class (e.g., a class implementing a stream) cannot initialize objects by copying the values from an existing object of that class (called *copy construction*) and cannot assign objects to each other.

Once a class defines at least one constructor its default constructor is not automatically defined by the compiler. The C++0x standard relaxes that restriction somewhat by offering the '`= default`' syntax. A class specifying '`= default`' with its default constructor declaration indicates that the trivial default constructor should be provided by the compiler. A trivial default constructor performs the following actions:

- Its data members of built-in or primitive types are not initialized;
- Its composed (class type) data memebrs are initialized by their default constructors.
- If the class is derived from a base class (cf. chapter 13) the base class is initialized by its default constructor.

Trivial implementations can also be provided for the *copy constructor* the *overloaded assignment operator* and the *destructor*, which are introduced in chapter 8.

Conversely, situations exist where some (otherwise automatically provided) members should *not* be made available. This is realized by specifying '`= delete`'. Using `= default` and `= delete` is illustrated by the following example. The default constructor receives its trivial implementation, copy-construction is prevented:

```
class Strings
{
    public:
        Strings() = default;
        Strings(std::string const *sp, size_t size);

        Strings(Strings const &other) = delete;
};
```

7.5 Const member functions and const objects

The keyword `const` is often used behind the parameter list of member functions. This keyword indicates that a member function does not alter the data members of its object. Such member functions are called *const member functions*. In the class `Person`, we see that the accessor functions were declared `const`:

```
class Person
{
    public:
        std::string const &name()    const;
        std::string const &address() const;
        std::string const &phone()   const;
        sie_t weight()               const;
};
```

The rule of thumb given in section 3.1.3 applies here too: whichever appears to the *left* of the keyword `const`, is not altered. With member functions this should be interpreted as 'doesn't alter its own data'.

When implementing a const member function its the `const` attribute must be repeated:

```
string const &Person::name() const
{
    return d_name;
}
```

The compiler will prevent the data members of a class from being modified by one of its const member functions. Therefore a staement like

```
d_name[0] = toupper(static_cast<unsigned char>(d_name[0]));
```

results in a compiler error when added to the above function's definition.

`Const` member functions useful to prevent inadvertent data modification. Except for constructors and the destructor (cf. chapter 8) only const member functions can be used with (plain, references or pointers to) `const` objects.

Const objects are frequently encounterd as `const &` parameters of functions. Inside such functions only the object's const members may be used. Here is an example:

```
void displayWeight(ostream &out, Person const &person)
{
    out << person.name() << " weighs " << person.weight() << " kg.\n";
}
```

Since `person` is defined as a `Person const &` the function `displayWeight` cannot call, e.g., `person.setWeight(75)`.

The `const` member function attribute can be used to overload member functions. When functions are overloaded by their `const` attribute the compiler will use the member function matching most closely the const-qualification of the object:

- When the object is a `const` object, only `const` member functions can be used.
- When the object is not a `const` object, non-`const` member functions will be used, *unless* only a `const` member function is available. In that case, the `const` member function will be used.

The next example illustrates how (non) `const` member functions are selected:

```
#include <iostream>
using namespace std;

class Members
{
    public:
        Members();
        void member();
        void member() const;
};

Members::Members()
{}
void Members::member()
{
    cout << "non const member\n";
```

```
}
void Members::member() const
{
    cout << "const member\n";
}

int main()
{
    Members const constObject;
    Members       nonConstObject;

    constObject.member();
    nonConstObject.member();
}
/*
        Generated output:

    const member
    non const member
*/
```

As a general principle of design: member functions should always be given the `const` attribute, unless they actually modify the object's data.

7.5.1 Anonymous objects

Sometimes objects are used because they offer a certain functionality. The objects only exist because of their functionality, and nothing in the objects themselves is ever changed. The following class `Print` offers a facility to print a string, using a configurable prefix and suffix. A partial class interface could be:

```
class Print
{
    public:
        Print(ostream &out);
        print(std::string const &prefix, std::string const &text,
                 std::string const &suffix) const;
};
```

An interface like this would allow us to do things like:

```
Print print(cout);
for (int idx = 0; idx < argc; ++idx)
    print.print("arg: ", argv[idx], "\n");
```

This works fine, but it could greatly be improved if we could pass `print`'s invariant arguments to `Print`'s constructor. This would simplify `print`'s prototype (only one argument would need to be passed rather than three) and we could wrap the above code in a function expecting a `Print` object:

```
void allArgs(Print const &print, int argc, char *argv[])
{
    for (int idx = 0; idx < argc; ++idx)
        print.print(argv[idx]);
}
```

The above is a fairly generic piece of code, at least it is with respect to `Print`. Since `prefix` and `suffix` don't change they can be passed to the constructor which could be given the prototype:

```
Print(ostream &out, string const &prefix = "", string const &suffix = "");
```

Now `allArgs` may be used as follows:

```
Print p1(cout, "arg: ", "\n");      // prints to cout
Print p2(cerr, "err: --", "--\n");  // prints to cerr

allArgs(p1, argc, argv);            // prints to cout
allArgs(p2, argc, argv);            // prints to cerr
```

But now we note that `p1` and `p2` are only used inside the `allArgs` function. Furthermore, as we can see from `print`'s prototype, `print` doesn't modify the internal data of the `Print` object it is using.

In such situations it is actually not necessary to define objects before they are used. Instead *anonymous objects* may be used. Anonymous objects can be used:

- to initialize a function parameter which is a `const` reference to an object;
- if the object is *only* used inside the function call.

Anonymous objects are defined when a constructor is used without providing a name for the constructed object. Here is the corresponding example:

```
allArgs(Print(cout, "arg: ", "\n"), argc, argv);    // prints to cout
allArgs(Print(cerr, "err: --", "--\n"), argc, argv);// prints to cerr
```

In this situation the `Print` objects are constructed and immediately passed as first arguments to the `printText` functions, where they are accessible as the function's `print` parameter. While the `allArgs` function is executing they can be used, but once the function has completed, the anonymous `Print` objects are no longer accessible.

7.5.1.1 Subtleties with anonymous objects

Anonymous objects can be used to initialize function parameters that are `const` references to objects. These objects are created just before such a function is called, and are destroyed once the function has terminated. **C++**'s grammar allows us to use anonymous objects in other situations as well. Consider the following snippet of code:

```
int main()
{
    // initial statements
    Print("hello", "world");
    // later statements
}
```

In this example an anonymous `Print` object is constructed, and it is immediately destroyed thereafter. So, following the 'initial statements' our `Print` object is constructed. Then it is destroyed again followed by the execution of the 'later statements'.

The example illustrates that the standard lifetime rules do not apply to anonymous objects. Their lifetimes are limited to the *statements*, rather than to the *end of the block* in which they are defined.

Plain anonymous object are at least useful in one situation. Assume we want to put *markers* in our code producing some output when the program's execution reaches a certain point. An object's constructor could be implemented so as to provide that marker-functionality allowing us to put markers in our code by defining anonymous, rather than named objects.

C++'s grammar contains another remarkable characteristic illustrated by the next example:

```
int main(int argc, char **argv)
{
    Print p(cout, "", "");              // 1
    allArgs(Print(p), argc, argv);      // 2
}
```

In this example a non-anonymous object `p` is constructed in statement 1, which is then used in statement 2 to *initialize* an anonymous object. The anonymous object, in turn, is then used to initialize `allArgs`'s `const` reference parameter. This use of an existing object to initialize another object is common practice, and is based on the existence of a so-called *copy constructor*. A copy constructor creates an object (as it is a constructor) using an existing object's characteristics to initialize the data of the object that's created. Copy constructors are discussed in depth in chapter 8, but presently only the concept of a copy constructor is used.

In the above example a copy constructor is used to initialize an anonymous object. The anonymous object was then used to initialize a parameter of a function. However, when we try to apply the same trick (i.e., using an existing object to initialize an anonymous object) to a plain statement, the compiler generates an error: the object `p` can't be redefined (in statement 3, below):

```
int main(int argc, char *argv[])
{
    Print p("", "");                    // 1
    allArgs(Print(p), argc, argv);      // 2
    Print(p);                           // 3 error!
}
```

Does this mean that using an existing object to initialize an anonymous object that is used as function argument is OK, while an existing object can't be used to initialize an anonymous object in a plain statement?

The compiler actually provides us with the answer to this apparent contradiction. About statement 3 the compiler reports something like:

```
error: redeclaration of 'Print p'
```

which solves the problem when realizing that within a compound statement objects and variables may be defined. Inside a compound statement, a *type name* followed by a `variable name` is the grammatical form of a variable definition. *Parentheses* can be used to break priorities, but if there are no priorities to break, they have no effect, and are simply ignored by the compiler. In statement 3 the parentheses allowed us to get rid of the blank that's required between a type name and the variable name, but to the compiler we wrote

```
    Print (p);
```

which is, since the parentheses are superfluous, equal to

```
    Print p;
```

thus producing `p`'s redeclaration.

As a further example: when we define a variable using a built-in type (e.g., `double`) using superfluous parentheses the compiler will quietly remove these parentheses for us:

```
double ((((a))));       // weird, but OK.
```

To summarize our findings about anonymous variables:

- Anonymous objects are great for initializing `const` reference parameters.
- The same syntaxis, however, can also be used in stand-alone statements, in which they are interpreted as variable definitions if our intention actually was to initialize an anonymous object using an existing object.
- Since this may cause confusion, it's probably best to restrict the use of anonymous objects to the first (and main) form: initializing function parameters.

7.6 The keyword 'inline'

Let us take another look at the implementation of the function `Person::name()`:

```
std::string const &Person::name() const
{
    return d_name;
}
```

This function is used to retrieve the name field of an object of the class `Person`. Example:

```
void showName(Person const &person)
{
    cout << person.name();
}
```

To insert `person`'s name the following actions are performed:

- The function `Person::name()` is called.
- This function returns `person`'s `d_name` as a reference.
- The referenced name is inserted into `cout`.

Especially the first part of these actions causes some time loss, since an extra function call is necessary to retrieve the value of the `name` field. Sometimes a faster procedure immediately making the `d_name` data member available without ever actually calling a function `name()`. This can be realized using `inline` functions. An inline function is a request to the compiler to insert the function's code at the location of the function's call. This may speed up execution by avoiding a function call, which typically comes with some (stack handling and parameter passing) overhead. Note that `inline` is a *request* to the compiler: the compiler may decide to ignore it, and *will* probably ignore it when the function's body contains much code. Good programming discipline suggests to be aware of this, and to avoid `inline` unless the function's body is fairly small. More on this in section 7.6.2.

7.6.1 Defining members inline

Inline functions may be implemented *in the class interface itself*. For the class `Person` this results in the following implementation of `name`:

```
class Person
{
    public:
        std::string const &name() const
```

```
        {
            return d_name;
        }
};
```

Note that the inline code of the function `name` now literally occurs inline in the interface of the class `Person`. The keyword `const` is again added to the function's header.

Although members can be defined *in-class* (i.e., inside the class interface itself), it is considered bad practice for the following reasons:

- Defining members inside the interface contaminates the interface with implementations. The interface's purpose is to document what functionality the class offers. Mixing member declarations and implementation details complicates understanding the interface. Readers need to skip implementation details which takes time and makes it hard to grab the 'broad picture', and thus to understand at a glance what functionality the class's objects are offering.
- In-class implementations of private member functions may usually be avoided altogether (as they are private members). They should be moved to the internal header file (*unless* inline public members use such inline private members).
- Although members that are eligible for inline-coding should remain inline, situations do exist where such inline members migrate from an inline to a non-inline definition. In-class inline definitions still need editing (sometimes considerable editing) before they can be compiled. This additional editing is undesirable.

Because of the above considerations inline members should not be defined in-class. Rather, they should be defined following the class interface. The `Person::name` member is therefore preferably defined as follows:

```
class Person
{
    public:
        std::string const &name() const;
};

inline std::string const &Person::name() const
{
    return d_name;
}
```

If it is ever necessary to cancel `Person::name`'s inline implementation, then this becomes it non-inline implementation:

```
#include "person.ih"

std::string const &Person::name() const
{
    return d_name;
}
```

Only the `inline` keyword needs to be removed to obtain the correct non-inline implementation.

Defining members inline has the following effect: whenever an inline-defined function is called, the compiler may *insert the function's body* at the location of the function call. It may be that the function itself is never actually called.

This construction, where the function code itself is inserted rather than a call to the function, is called an inline function. Note that using inline functions may result in multiple occurrences of the code of

those functions in a program: one copy for each invocation of the inline function. This is probably OK if the function is a small one, and needs to be executed fast. It's not so desirable if the code of the function is extensive. The compiler knows this too, and handles the use of inline functions as a *request* rather than a *command*. If the compiler considers the function too long, it will not grant the request. Instead it will treat the function as a normal function.

7.6.2 When to use inline functions

When should inline functions be used, and when not? There are some rules of thumb which may be followed:

- In general inline functions should **not** be used. *Voilà*; that's simple, isn't it?
- Consider defining a function inline once a fully developed and tested program runs too slowly and shows 'bottlenecks' in certain functions, and the bottleneck is removed by defining inline members. A profiler, which runs a program and determines where most of the time is spent, is necessary to perform for such optimizations.
- Defining an inline functions may be considered when they consist of one very simple statement (such as the return statement in the function `Person::name`).
- When a function is defined inline, its implementation is inserted in the code wherever the function is used. As a consequence, when the *implementation* of the inline function changes, all sources using the inline function must be recompiled. In practice that means that all functions must be recompiled that include (either directly or indirectly) the header file of the class in which the inline function is defined. Not a very attractive prospect.
- It is only useful to implement an inline function when the time spent during a function call is long compared to the time spent by the function's body. An example of an inline function which will hardly have any effect on the program's speed is:

  ```
  inline void Person::printname() const
  {
      cout << d_name << endl;
  }
  ```

  This function contains only one statement. However, the statement takes a relatively long time to execute. In general, functions which perform input and output take lots of time. The effect of the conversion of this function `printname()` to inline would therefore lead to an insignificant gain in execution time.

All inline functions have one disadvantage: the actual code is inserted by the compiler and must therefore be known compile-time. Therefore, as mentioned earlier, an inline function can never be located in a run-time library. Practically this means that an inline function is found near the interface of a class, usually in the same header file. The result is a header file which not only shows the **declaration** of a class, but also part of its **implementation**, thus always blurring the distinction between interface and implementation.

7.7 Local classes: classes inside functions

Classes are usually defined at the global or namespace level. However, it is entirely possible to define a local class, i.e., inside a function. Such classes are called *local classlocal classes*.

Local classes can be very useful in advanced applications involving inheritance or templates (cf. section 13.8). At this point in the **C++** Annotations they have limited use, although their main features can be described. At the end of this section an example is provided.

- Local classes may use almost all characteristics of normal classes. They may have constructors, destructors, data members, and member functions;
- Local classes cannot define static data members. Static member functions, however, *can* be defined.
- Since a local class may define static member functions, it is possible to define *nested functions* in **C++** somewhat comparable to the way programming languages like **Pascal** allow nested functions to be defined.
- If a local class needs access to a constant integral value, a local *enum* can be used. The `enum` may be anonymous, exposing only the `enum` values.
- Local classes cannot directly access the non-static variables of their surrounding context. For example, in the example shown below the class `Local` cannot directly access `main`'s `argc` parameter.
- Local classes may directly access global data and static variables defined by their surrounding function. This includes variables defined in the anonymous namespace of the source file containing the local class.
- Local class objects can be defined inside the function body, but they cannot leave the function as objects of their own type. I.e., a local class name cannot be used for either the return type or for the parameter types of its surrounding function.
- As a prelude to *inheritance* (chapter 13): a local class may be derived from an existing class allowing the surrounding function to return a dynamically allocated locally constructed class object, pointer or reference could be returned via a base class pointer or reference.

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char *argv[])
{
    static size_t staticValue = 0;

    class Local
    {
        int d_argc;             // non-static data members OK

        public:
            enum                // enums OK
            {
                value = 5
            };
            Local(int argc)     // constructors and member functions OK
            :                   // in-class implementation required
                d_argc(argc)
            {
                                // global data: accessible
                cout << "Local constructor\n";
                                // static function variables: accessible
                staticValue += 5;
            }
            static void hello() // static member functions: OK
            {
                cout << "hello world\n";
            }
    };
```

```
    Local::hello();             // call Local static member
    Local loc(argc);            // define object of a local class.
}
```

7.8 The keyword 'mutable'

Earlier, in section 7.5, the concepts of const member functions and const objects were introduced.

C++ also allows the declaration of data members which may be modified, even by const member function. The declaration of such data members in the class interface start with the keyword `mutable`.

Mutable should be used for those data members that may be modified without logically changing the object, which might therefore still be considered a constant object.

An example of a situation where `mutable` is appropriately used is found in the implementation of a string class. Consider the `std::string`'s `c_str` and `data` members. The actual data returned by the two members are identical, but `c_str` must ensure that the returned string is terminated by an ASCII-Z byte. As a string object has both a length and a capacity there an easy way to implement `c_str` is to ensure that the string's capacity is at least one larger than its length. This invariant allows `c_str` to be implemented as follows:

```
char const *string::c_str() const
{
    d_data[d_length] = 0;
    return d_data;
}
```

This implementation logically does not modify the object's data as the bytes beyond the object's initial (length) characters have undefined values. But in order to use this implementation `d_data` must be declared `mutable`:

```
mutable char *d_data;
```

The keyword `mutable` is also useful in classes implementing, e.g., reference counting. Consider a class implementing reference counting for textstrings. The object doing the reference counting might be a const object, but the class may define a copy constructor. Since const objects can't be modified, how would the copy constructor be able to increment the reference count? Here the `mutable` keyword may profitably be used, as it can be incremented and decremented, even though its object is a const object.

The keyword `mutable` should sparingly be used. Data modified by const member functions should never logically modify the object, and it should be easy to demonstrate this. As a rule of thumb: do not use `mutable` unless there is a very clear reason (the object is logically not altered) for violating this rule.

7.9 Header file organization

In section 2.5.10 the requirements for header files when a **C++** program also uses **C** functions were discussed. Header files containing class interfaces have additional requirements.

First, source files. With the exception of the occasional classless function, source files contain the code of member functions of classes. here there are basically two approaches:

- All required header files for a member function are included in each individual source file.
- All required header files (for all member functions of a class) are included in a header file that is included by each of the source files defining class members.

The first alternative has the advantage of economy for the compiler: it only needs to read the header files that are necessary for a particular source file. It has the disadvantage that the program developer must include multiple header files again and again in sourcefiles: it both takes time to type the `include`-directives and to think about the header files which are needed in a particular source file.

The second alternative has the advantage of economy for the program developer: the header file of the class accumulates header files, so it tends to become more and more generally useful. It has the disadvantage that the compiler frequently has to process many header files which aren't actually used by the function to compile.

With computers running faster and faster (and compilers getting smarter and smarter) I think the second alternative is to be preferred over the first alternative. So, as a starting point source files of a particular class `MyClass` could be organized according to the following example:

```
#include <myclass.h>

int MyClass::aMemberFunction()
{}
```

There is only one `include`-directive. Note that the directive refers to a header file in a directory mentioned in the `INCLUDE`-file environment variable. Local header files (using `#include "myclass.h"`) could be used too, but that tends to complicate the organization of the class header file itself somewhat.

The organization of the header file itself requires some attention. Consider the following example, in which two classes `File` and `String` are used.

Assume the `File` class has a member `gets(String &destination)`, while the class `String` has a member function `getLine(File &file)`. The (partial) header file for the `class String` is then:

```
#ifndef STRING_H_
#define STRING_H_

#include <project/file.h>   // to know about a File

class String
{
    public:
        void getLine(File &file);
};
#endif
```

Unfortunately a similar setup is required for the class `File`:

```
#ifndef FILE_H_
#define FILE_H_

#include <project/string.h>   // to know about a String

class File
{
    public:
        void gets(String &string);
};
#endif
```

Now we have created a problem. The compiler, trying to compile the source file of the function `File::gets()` proceeds as follows:

- The header file `project/file.h` is opened to be read;

- `FILE_H_` is defined
- The header file `project/string.h` is opened to be read
- `STRING_H_` is defined
- The header file `project/file.h` is (again) opened to be read
- Apparently, `FILE_H_` is already defined, so the remainder of `project/file.h` is skipped.
- The interface of the class `String` is now parsed.
- In the class interface a reference to a `File` object is encountered.
- As the `class File` hasn't been parsed yet, a `File` is still an undefined type, and the compiler quits with an error.

The solution to this problem is to use a *forward class reference before* the class interface, and to include the corresponding class header file *beyond* the class interface. So we get:

```
#ifndef STRING_H_
#define STRING_H_

class File;                 // forward reference

class String
{
    public:
        void getLine(File &file);
};

#include <project/file.h>   // to know about a File

#endif
```

A similar setup is required for the class `File`:

```
#ifndef FILE_H_
#define FILE_H_

class String;               // forward reference

class File
{
    public:
        void gets(String &string);
};

#include <project/string.h>   // to know about a String

#endif
```

This works well in all situations where either references or pointers to other classes are involved and with (non-inline) member functions having class-type return values or parameters.

This setup doesn't work with composition, nor with in-class inline member functions. Assume the class `File` has a *composed* data member of the class `String`. In that case, the class interface of the class `File` *must* include the header file of the class `String` before the class interface itself, because otherwise the compiler can't tell how big a `File` object will be, as it doesn't know the size of a `String` object once the interface of the `File` class is completed.

In cases where classes contain composed objects (or are derived from other classes, see chapter 13) the header files of the classes of the composed objects must have been read *before* the class interface itself. In such a case the `class File` might be defined as follows:

```
#ifndef FILE_H_
#define FILE_H_

#include <project/string.h>     // to know about a String

class File
{
    String d_line;              // composition !

    public:
        void gets(String &string);
};
#endif
```

The class `String` can't declare a `File` object as a composed member: such a situation would again result in an undefined class while compiling the sources of these classes.

All remaining header files (appearing below the class interface itself) are required only because they are used by the class's source files.

This approach allows us to introduce yet another refinement:

- Header files defining a class interface should *declare* what can be declared before defining the class interface itself. So, classes that are mentioned in a class interface should be specified using forward declarations *unless*
  - They are a *base class* of the current class (see chapter 13);
  - They are the class types of composed data members;
  - They are used in inline member functions.

  In particular: additional actual header files are *not* required for:
  - class-type return values of functions;
  - class-type value parameters of functions.

  Class header files of objects that are either composed or inherited or that are used in inline functions, *must* be known to the compiler before the interface of the current class starts. The information in the header file itself is protected by the `#ifndef ... #endif` construction introduced in section 2.5.10.
- Program sources in which the class is used only need to include this header file. *Lakos*, (2001) refines this process even further. See his book **Large-Scale C++ Software Design** for further details. This header file should be made available in a well-known location, such as a directory or subdirectory of the standard `INCLUDE` path.
- To implementat member functions the class's header file is required and usually additional header files (like the `string` header file) as well. The class header file itself as well as these additional header files should be included in a separate internal header file (for which the extension `.ih` ('internal header') is suggested).

  The `.ih` file should be defined in the same directory as the source files of the class. It has the following characteristics:
  - There is *no* need for a protective `#ifndef .. #endif` shield, as the header file is never included by other header files.
  - The standard `.h` header file defining the class interface is included.

- The header files of all classes used as forward references in the standard `.h` header file are included.
- Finally, all other header files that are required in the source files of the class are included.

An example of such a header file organization is:

- First part, e.g., `/usr/local/include/myheaders/file.h`:

```
#ifndef FILE_H_
#define FILE_H_

#include <fstream>      // for composed 'ifstream'

class Buffer;           // forward reference

class File              // class interface
{
    std::ifstream d_instream;

    public:
        void gets(Buffer &buffer);
};
#endif
```

- Second part, e.g., `~/myproject/file/file.ih`, where all sources of the class File are stored:

```
#include <myheaders/file.h> // make the class File known

#include <buffer.h>         // make Buffer known to File
#include <string>           // used by members of the class
#include <sys/stat.h>       // File.
```

7.9.1 Using namespaces in header files

When entities from namespaces are used in header files, no `using` directive should be specified in those header files if they are to be used as general header files declaring classes or other entities from a library. When the `using` directive is used in a header file then users of such a header file are forced to accept and use the declarations in all code that includes the particular header file.

For example, if in a namespace `special` an object `Inserter cout` is declared, then `special::cout` is of course a different object than `std::cout`. Now, if a class `Flaw` is constructed, in which the constructor expects a reference to a `special::Inserter`, then the class should be constructed as follows:

```
class special::Inserter;

class Flaw
{
public:
    Flaw(special::Inserter &ins);
};
```

Now the person designing the class `Flaw` may be in a lazy mood, and might get bored by continuously having to prefix `special::` before every entity from that namespace. So, the following construction is used:

```
using namespace special;
```

```
class Inserter;

class Flaw
{
public:
    Flaw(Inserter &ins);
};
```

This works fine, up to the point where somebody wants to include `flaw.h` in other source files: because of the `using` directive, this latter person is now by implication also `using namespace special`, which could produce unwanted or unexpected effects:

```
#include <flaw.h>
#include <iostream>

using std::cout;

int main()
{
    cout << "starting" << endl;     // doesn't compile
}
```

The compiler is confronted with two interpretations for `cout`: first, because of the `using` directive in the `flaw.h` header file, it considers `cout` a `special::Inserter`, then, because of the `using` directive in the user program, it considers `cout` a `std::ostream`. Consequently, the compiler reports an error.

As a rule of thumb, header files intented for general use should not contain `using` declarations. This rule does not hold true for header files which are only included by the sources of a class: here the programmer is free to apply as many `using` declarations as desired, as these directives never reach other sources.

7.10 Sizeof applied to class data members (C++0x, 4.4)

In the C++0x standard the `sizeof` operator can be applied to data members of classes without the need to specify an object as well. Consider:

```
class Data
{
    std::string d_name;
    ...
};
```

To obtain the size of `Data`'s `d_name` member C++0x allows the following expression:

```
sizeof(Data::d_name);
```

However, note that the compiler observes data protection here as well. `Sizeof(Data::d_name)` can only be used where `d_name` may be visible as well, i.e., by `Data`'s member functions and friends.

7.11 Unrestricted Unions (C++0x, ?)

The C++0x standard allows unions to be defined consisting of objects for which a non-trivial constructor is defined. The example illustrates how such an *unrestricted union* may be defined. The data field `u_string` is an object of the class `std::string` for which non-trivial constructors are available:

```
union Union
{
    int u_int;
    double u_double;
    std::string u_string;
};
```

Chapter 8

Classes And Memory Allocation

In contrast to the set of functions that handle memory allocation in **C** (i.e., `malloc` etc.), memory allocation in **C++** is handled by the operators `new` and `delete`. Important differences between `malloc` and `new` are:

- The function `malloc` doesn't 'know' what the allocated memory will be used for. E.g., when memory for `int`s is allocated, the programmer must supply the correct expression using a multiplication by `sizeof(int)`. In contrast, `new` requires a type to be specified; the `sizeof` expression is implicitly handled by the compiler. Using `new` is therefore *type safe*.
- Memory allocated by `malloc` is initialized by `calloc`, initializing the allocated characters to a configurable initial value. This is not very useful when objects are available. As operator `new` knows about the type of the allocated entity it may (and will) call the constructor of an allocated class type object. This constructor may be also supplied with arguments.
- All **C**-allocation functions must be inspected for `NULL`-returns. This is not required anymore when `new` is used. In fact, `new`'s behavior when confronted with failing memory allocation is configurable through the use of a *new_handler* (cf. section 8.2.2).

A comparable relationship exists between `free` and `delete`: `delete` makes sure that when an object is deallocated, its destructor is automatically called.

The automatic calling of constructors and destructors when objects are created and destroyed has consequences which we shall discuss in this chapter. Many problems encountered during **C** program development are caused by incorrect memory allocation or memory leaks: memory is not allocated, not freed, not initialized, boundaries are overwritten, etc.. **C++** does not 'magically' solve these problems, but it *does* provide us with tools to prevent these kinds of problems.

As a consequence of `malloc` and friends becoming deprecated the very frequently used `str...` functions, like `strdup`, that are all `malloc` based, should be avoided in **C++** programs. Instead, the facilities of the `string` class and operators `new` and `delete` should be used instead.

Memory allocation procedures influence the way classes dynamically allocating their own memory should be designed. Therefore, in this chapter these topics are discussed in addition to discussions about operators `new` and `delete`. We'll first cover the peculiarities of operators `new` and `delete`, followed by a discussion about:

- the destructor: the member function that's called when an object ceases to exist;
- the assignment operator, allowing us to assign an object to another object of its own class;
- the `this` pointer, allowing explicit references to the object for which a member function was called;
- the copy constructor: the constructor creating a copy of an object;
- the move constructor: a constructor creating an object from an anonymous temporary object.

8.1 Operators 'new' and 'delete'

C++ defines two operators to allocate memory and to return it to the 'common pool'. These operators are, respectively `new` and `delete`.

Here is a simple example illistrating their use. An `int` pointer variable points to memory allocated by operator `new`. This memory is later released by operator `delete`.

```
int *ip = new int;
delete ip;
```

Here are some characteristics of operators `new` and `delete`:

- `new` and `delete` are *operators* and therefore do not require parentheses, as required for *functions* like `malloc` and `free`;
- `new` returns a pointer to the kind of memory that's asked for by its operand (e.g., it returns a pointer to an `int`);
- `new` uses a *type* as its operand, which has the important benefit that the correct amount of memory, given the type of the object to be allocated, is made available;
- as a consequence, `new` is a type safe operator as it always returns a pointer to the type that was mentioned as its operand. In addition, the type of the receving pointer must match the type specified with operator `new`;
- `new` may fail, but this is normally of *no* concern to the programmer. In particular, the program does *not* have to test the success of the memory allocation, as is required for `malloc` and friends. Section 8.2.2 delves into this aspect of `new`;
- `delete` returns `void`;
- for each call to `new` a matching `delete` should eventually be executed, lest a memory leak occurs;
- `delete` can safely operate on a 0-pointer (doing nothing);
- otherwise `delete` must only be used to return memory allocated by `new`. It should *not* be used to return memory allocated by `malloc` and friends.
- in **C++** malloc and friends are *deprecated* and should be avoided.

Operator `new` can be used to allocate primitive types but also to allocate objects. When a primitive type or a `struct` type without a constructor is allocated the allocated memory is *not* guaranteed to be initialized to 0, but an initialization expression may be provided:

```
int *v1 = new int;          // not guaranteed to be initialized to 0
int *v1 = new int();        // initialized to 0
int *v2 = new int(3);       // initialized to 3
int *v3 = new int(3 * *v2); // initialized to 9
```

When a class-type object is allocated, the arguments of its constructor (if any) are specified immediately following the type specification in the `new` expression and the object will be initialized according to the thus specified constructor. For example, to allocate `string` objects the following statements could be used:

```
string *s1 = new string;           // uses the default constructor
string *s2 = new string();         // same
string *s3 = new string(4, ' ');   // initializes to 4 blanks.
```

In addition to using `new` to allocate memory for a single entity or an array of entities there is also a variant that allocates *raw memory*: `operator new(sizeInBytes)`. Raw memory is returned as a `void *`. Here `new` allocates a block of memory for unspecified purpose. Although raw memory may consist of multiple characters it should not be interpreted as an array of characters. Since raw memory returned by `new` is returned as a `void *` its return value can be assigned to a `void *` variable. More often it is assigned to a `char *` variable, using a cast. Here is an example:

```
char *chPtr = static_cast<char *>(operator new(numberOfBytes));
```

The use of raw memory is frequently encounted in combination with the *placement new* operator, discussed in section 8.1.4.

8.1.1 Allocating arrays

Operator `new[]` is used to allocate arrays. The generic notation `new[]` is used in the **C++** Annotations. Actually, the number of elements to be allocated must be specified between the square brackets and it must, in turn, be *prefixed* by the type of the entities that must be allocated. Example:

```
int *intarr = new int[20];          // allocates 20 ints
string *stringarr = new string[10]; // allocates 10 strings.
```

Operator `new` is a different operator than operator `new[]`. A consequence of this difference is discussed in the next section (8.1.2).

Arrays allocated by operator `new[]` are called *dynamic arrays*. They are constructed during the execution of a program, and their lifetime may exceed the lifetime of the function in which they were created. Dynamically allocated arrays may last for as long as the program runs.

When `new[]` is used to allocate an array of primitive values or an array of objects, `new[]` must be specified with a type and an (unsigned) expression between its square brackets. The type and expression together are used by the compiler to determine the required size of the block of memory to make available. When `new[]` is used the array's elements are stored consecutively in memory. An array index expression may thereafter be used to access the array's individual elements: `intarr[0]` represents the first `int` value, immediately followed by `intarr[1]`, and so on until the last element (`intarr[19]`). With non-class types (primitive types, `struct` types without constructors) the block of memory returned by operator `new[]` is *not* guaranteed to be initialized to 0.

When operator `new[]` is used to allocate arrays of objects their constructors are automatically used. Consequently `new string[20]` results in a block of 20 *initialized* `string` objects. When allocating arrays of objects the class's *default constructor* is used to initialize each individual object in turn. A non-default constructor cannot be called, but often it is possible to work around that as discussed in section 13.8.

The expression between brackets of operator `new[]` represents the number of elements of the array to allocate. The **C++** standard allows allocation of 0-sized arrays. The statement `new int[0]` is correct **C++**. However, it is also pointless and confusing and should be avoided. It is pointless as it doesn't refer to any element at all, it is confusing as the returned pointer has a useless non-0 value. A pointer intending to point to an array of values should be initialized (like any pointer that isn't yet pointing to memory) to 0, allowing for expressions like `if (ptr) ...`

Without using operator `new[]`, arrays of variable sizes can also be constructed as *local arrays*. Such arrays are not dynamic arrays and their lifetimes are restricted to the lifetime of the block in which they were defined.

> Once allocated, all arrays have fixed sizes. There is no simple way to enlarge or shrink arrays. **C++** has no operator '`renew`'. Section 8.1.3 illustrates how an array can be enlarged.

8.1.2 Deleting arrays

Dynamically allocated arrays are deleted using operator `delete[]`. It expects a pointer to a block of memory, previously allocated by operator `new[]`.

When operator `delete[]`'s operand is a pointer to an array of objects two actions will be performed:

- First, the class's destructor is called for each of the objects in the array. The destructor, as explained later in this chapter, performs all kinds of cleanup operations that are required by the time the object ceases to exist.
- Second, the memory pointed at by the pointer is returned to the common pool.

Here is an example showing how to allocate and delete an array of 10 string objects:

```
std::string *sp = new std::string[10];
delete[] sp;
```

No special action is performed if a dynamically allocated array of primitive typed values is deleted. Following `int *it = new int[10]` the statement `delete[] it` simply returns the memory pointed at by `it` is returned. Realize that, as a pointer is a primitive type, deleting a dynamically allocated array of pointers to objects will *not* result in the proper destruction of the objects the array's elements point at. So, the following example results in a *memory leak*:

```
string **sp = new string *[5];
for (size_t idx = 0; idx != 5; ++idx)
    sp[idx] = new string;
delete[] sp;            // MEMORY LEAK !
```

In this example the only action performed by `delete][` is to return an area the size of five pointers to strings to the common pool.

Here's how the destruction in such cases *should* be performed:

- Call `delete` for each of the array's elements;
- Delete the array itself

Example:

```
for (size_t idx = 0; idx != 5; ++idx)
    delete sp[idx];
delete[] sp;
```

One of the consequences is of course that by the time the memory is going to be returned not only the pointer must be available but also the number of elements it contains. This can easily be accomplished by storing pointer and number of elements in a simple class and then using an object of that class.

Operator `delete[]` is a different operator than operator `delete`. The rule of thumb is: if `new[]` was used, also use `delete[]`.

8.1.3 Enlarging arrays

Once allocated, all arrays have fixed sizes. There is no simple way to enlarge or shrink arrays. **C++** has no `renew` operator. The basic steps to take when enlarging an array are the following:

- Allocate a new block of memory of larger size;

- Copy the old array contents to the new array;
- Delete the old array;
- Let the pointer to the array point to the newly allocated array.

Static and local arrays cannot be resized. Resizing is only possible for dynamically allocated arrays. Example:

```
#include <string>
using namespace std;

string *enlarge(string *old, unsigned oldsize, unsigned newsize)
{
    string *tmp = new string[newsize];  // allocate larger array

    for (size_t idx = 0; idx != oldsize; ++idx)
        tmp[idx] = old[idx];            // copy old to tmp

    delete[] old;                       // delete the old array
    return tmp;                         // return new array
}

int main()
{
    string *arr = new string[4];        // initially: array of 4 strings
    arr = enlarge(arr, 4, 6);           // enlarge arr to 6 elements.
}
```

The procedure to enlarge shown in the example also has several drawbacks.

- The new array requires `newsize` constructors to be called;
- Having initialized the strings in the new array, `oldsize` of them are immediately reassigned to the corresponding values in the original array;
- All the objects in the old arrays are destroyed.

Depending on the context various solutions exist to improve the efficiency of this rather inefficent procedure. An array of pointers could be used (requiring only the pointers to be copied, no destruction, no superfluous initialization) or raw memory in combination with the placement new operator could be used (an array of objects remains available, no destruction, no superfluous construction).

8.1.4 The 'placement new' operator

A remarkable form of operator `new` is called the *placement new* operator. Here operator `new` is provided with an existing block of memory in which an object or value is initialized. The block of memory should of course be large enough to contain the object, but apart from that no other requirements exist. It is easy to determine how much memory is used by en entity (object or variable) of type `Type`: the `sizeof` operator returns the number of bytes required by an `Type` entity. Entities may of course dynamically allocate memory for their own use. Dynamically allocated memory, however, is not part of the entity's memory 'footprint' but it is always made available externally to the entity itself. This is why `sizeof` returns the same value when applied to different `string` objects returning different length and capacity values.

The placement `new` operator uses the following syntax (using `Type` to indicate the used data type):

```
Type *new(void *memory) Type(arguments);
```

Here, memory is block of memory of at least sizeof(Type) bytes large and Type(arguments) is any constructor of the class Type.

The placement new operator is useful in situations where classes set aside memory to be used later. This is used, e.g., by std::string to change its capacity. Calling string::reserve may enlarge that capacity without making memory beyond the string's length immediately available. But the object itself may access its additional memory and so when information is added to a string object it can draw memory from its capacity rather than having to perform a reallocation for each single addition of information.

Let's apply that philosophy to a class Strings storing std::string objects. The class defines a char *d_memory accessing the memory holding its d_size string objects as well as d_capacity - d_size reserved memory. Assuming that a default constructor initializes d_capacity to 1, doubling d_capacity whenever an additional string must be stored, the class must support the following essential operations:

- doubling its capacity when all its reserve memory has been consumed;
- adding another string object
- properly deleting the installed strings and memory when a Strings object ceases to exist.

To double the capacity new memory is allocated, old memory is copied into the newly allocated memory, and the old memory is deleted. This is implemented by the member void Strings::reserve, assuming d_capacity has already been given its proper value:

```
void Strings::reserve()
{
    char *newMemory =
        static_cast<char *>(memcpy(
                                operator new(d_capacity),
                                d_memory,
                                d_size * sizeof(std::string)
                            ));

    delete d_memory;
    d_memory = newMemory;
}
```

The raw memory is made available by operator new(sizeInBytes). This should not be interpreted as an array of any kind, so a plain delete d_memory is used to return the previously allocated block of raw memory.

The member append adds another string object to a Strings object. A (public) member reserve(request) ensures that the String object's capacity is sufficient. Then the placement new operator is used to install the next string into the raw memory's appropriate location:

```
void Strings::append(std::string const &next)
{
    reserve(d_size + 1);

    new (reinterpret_cast<std::string *>(d_memory) + d_size)
        std::string(next);

    ++d_size;
}
```

At the end of the String object's lifetime all its dynamically allocated memory must be returned. This is the responsibility of the destructor, as explained in the next section. The destructor's full definition is postponed to that section, but its actions when placement new is involved can be discussed here.

With placement new an interesting situation is encountered. Objects, possibly themselves allocating memory, are installed in memory that may or may not have been allocated dynamically, but that is definitely not completely filled with such objects. So a simple delete[] can't be used, but a delete for each of the objects that *are* available can't be used either, since that would also delete the memory of the objects themselves, which wasn't dynamically allocated.

This peculiar situation is solved in a peculiar way, only encountered in cases where the placement new operator has been used: memory allocated by objects initialized using placement new is returned by *explicitly* calling the object's destructor. The destructor is declared as a member having the class preceded by a tilde as its name, not using any arguments. So, std::string's destructor is named ~string. The memory allocated by our class Strings is therefore properly destroyed as follows (in the example assume that using namespace std was specfied):

```
for
(
    string *sp = reinterpret_cast<string *>(d_memory) + d_size;
        sp-- != reinterpret_cast<string *>(d_memory);
)
    sp->~string();

delete d_memory;
```

So far, so good. All is well as long as we're using but one object. What about allocating an array of objects? Initialization is performed as usual. But as with delete, delete[] cannot be called when the buffer was allocated statically. Instead, when multiple objects were initialized using the placement new operator in combination with a statically allocated buffer all the objects' destructors must be called explicitly, as in the following example:

```
char buffer[3 * sizeof(string)];
string *sp = new(buffer) string [3];

for (size_t idx = 0; idx < 3; ++idx)
    sp[idx].~string();
```

8.2 The destructor

Comparable to the constructor, classes may define a *destructor*. This function is the constructor's counterpart in the sense that it is invoked when an object ceases to exist. A destructor is usually called automatically, but that's not always true. The destructors of dynamically allocated objects are not automatically activated, but in addition to that: when a program is interrupted by an exit call, only the destructors of already initialized global objects are called. In that situation destructors of objects defined *locally* by functions are also *not* called. This is one (good) reason for avoiding exit in **C++** programs.

Destructors obey the following syntactical requirements:

- a destructor's name is equal to its class name prefixed by a tilde;
- a destructor has no arguments;
- a destructor has no return value.

Destructors are declared in their class interfaces. Example:

```
class StringStore
{
```

```
    public:
        StringStore();
        ~StringStore();     // the destructor
};
```

By convention the constructors are declared first. The destructor is declared next, to be followed by other member functions.

A destructor's main task is to ensure that memory allocated by an object is properly returned when the object ceases to exist. Consider the following interface of the class `StringStore`:

```
class StringStore
{
    std::string *d_string;
    size_t d_size;

    public:
        StringStore();
        StringStore(char const *const *cStrings, size_t n);
        ~StringStore();

        std::string const &at(size_t idx) const;
        size_t size() const;
};
```

The constructor's task is to initialize the data fields of the object. E.g, its constructors are defined as follows:

```
StringStore::StringStore()
:
    d_string(0),
    d_size(0)
{}

StringStore::StringStore(char const *const *cStrings, size_t size)
:
    d_string(new string[size]),
    d_size(size)
{
    for (size_t idx = 0; idx != size; ++idx)
        d_string[idx] = cStrings[idx];
}
```

As objects of the class `StringStore` allocate memory a destructor is clearly required. Destructors may or may not be called automatically. Here are the rules:

- A destructor is *never* called unless its constructor completed successfully. The remaining rules only apply when this rule holds true;
- Destructors of local non-static objects are called automatically when the execution flow leaves the block in which they were defined; the destructors of objects defined somewhere in the outer block of a function are called just before the function terminates.
- Destructors of static or global objects are called when the program itself terminates.
- The destructor of a dynamically allocated object is called by `delete` using the object's address as its operand;
- The destructors of a dynamically allocated array of objects are called by `delete[]` using the address of the array's first element as its operand;

- The destructor of an object initialized by placement `new` is activated by explicitly calling the object's destructor.

The destructor's task is to ensure that all memory that is dynamically allocated and controlled only by the object itself is returned. The task of the `StringStore`'s destructor would therefore be to delete the memory to which `d_string` points. Its implementation is:

```
StringStore::~StringStore()
{
    delete[] d_string;
}
```

The next example shows `StringStore` at work. In `process` a `StringStore store` is created, and its data are displayed. It returns a dynamically allocated `StringStore` object to `main`. A `StringStore *` receives the address of the allocated object and deletes the object again. Another `StringStore` object is then created in a block of memory made available locally in `main`, and an explicit call to `~StringStore` is required to return the memory allocated by that object. In the example only once a `StringStore` object is automatically destroyed: the local `StringStore` object defined by `display`. The other two `StringStore` objects require explicit actions to prevent memory leaks.

```
#include "stringstore.h"
#include <iostream>

using namespace std;;

void display(StringStore const &store)
{
    for (size_t idx = 0; idx != store.size(); ++idx)
        cout << store.at(idx) << '\n';
}

StringStore *process(char *argv[], int argc)
{
    StringStore store(argv, argc);
    display(store);
    return new StringStore(argv, argc);
}

int main(int argc, char *argv[])
{
    StringStore *sp = process(argv, argc);
    delete sp;

    char buffer[sizeof(StringStore)];
    sp = new (buffer) StringStore(argv, argc);
    sp->~StringStore();
}
```

8.2.1 Object pointers revisited

Operators `new` and `delete` are used when an object or variable is allocated. One of the advantages of the operators `new` and `delete` over functions like `malloc` and `free` is that `new` and `delete` call the corresponding object constructors and destructors.

The allocation of an object by operator `new` is a two-step process. First the memory for the object itself is allocated. Then its constructor is called, initializing the object. Aanalogously to the construction of

an object, the destruction is also a two-step process: first, the destructor of the class is called deleting the memory controlled by the object. Then the memory used by the object itself is freed.

Dynamically allocated arrays of objects can also be handled by `new` and `delete`. When allocating an array of objects using operator `new` the default constructor is called for each object in the array. In cases like this operator `delete[]` must be used to ensure that the destructor is called for each of the objects in array.

Hoewever, the addresses returned by `new Type` and `new Type[size]` are of identical types, in both cases a `Type *`. Consequently it cannot be determined by the type of the pointer whether a pointer to dynamically allocated memory points to a single entity or to an array of entities.

What happens if `delete` rather than `delete[]` is used? Consider the following situation, in which the destructor `~StringStore` is modified so that it tells us that it is called. In a `main` function an array of two `StringStore` objects is allocated by `new`, to be deleted by `delete []`. Next, the same actions are repeated, albeit that the `delete` operator is called without `[]`:

```
    #include <iostream>
    #include "stringstore.h"
    using namespace std;

    StringStore::~StringStore()
    {
        cout << "StringStore destructor called" << '\n';
    }

    int main()
    {
        StringStore *a  = new StringStore[2];

        cout << "Destruction with []'s" << '\n';
        delete[] a;

        a = new StringStore[2];

        cout << "Destruction without []'s" << '\n';
        delete a;
    }
/*
    Generated output:
Destruction with []'s
StringStore destructor called
StringStore destructor called
Destruction without []'s
StringStore destructor called
*/
```

From the generated output, we see that the destructors of the individual `StringStore` objects are called when `delete[]` is used, while only the first object's destructor is called if the `[]` is omitted.

Conversely, if `delete[]` is called in a situation where `delete` should have been called the results are unpredicable, and will most likely cause the program to crash. This problematic behavior is caused by the way the run-time system stores information about the size of the allocated array (usually right *before* the array's first element). If a single object is allocated the array-specific information is not available, but it is nevertheless assumed present by `delete[]`. This latter operator will interpret bogus values before the array's first element as size information, thus usually causing the program to fail.

If no destructor is defined, a *trivial destructor* is defined by the compiler. The trivial destructor ensures that the destructors of composed objects (as well as the destructors of *base classes* if a class is a

derived class, cf. chapter 13) are called. This has serious implications: objects allocating memory will cause a memory leak unless precautionary measures are taken (by defining an appropriate destructor). Consider the following program:

```
#include <iostream>
#include "stringstore.h"
using namespace std;

StringStore::~StringStore()
{
    cout << "StringStore destructor called" << '\n';
}

int main()
{
    StringStore **ptr = new StringStore* [2];

    ptr[0] = new StringStore[2];
    ptr[1] = new StringStore[2];

    delete[] ptr;
}
```

This program produces no output at all. Why is this? The variable `ptr` is defined as a pointer to a pointer. The dynamically allocated array therefore consists of pointer variables and pointers are of a primitive type. No destructors exist for primitive typed variables. Consequently only the array itself is returned, and no `StringStore` destructor is called.

Of course, we don't want this, but require the `StringStore` objects pointed to by the elements of `a` to be deleted too. In this case we have two options:

- In a for-statement visit all the elements of the `ptr` array, calling `delete` for each of the array's elements. This procedure was demonstrated in the previous section.
- A wrapper class is designed around a pointer (to, e.g., an object of some class, like `StringStore`). Rather than using a pointer to a pointer to `StringStore` objects a pointer to an array of wrapper-class objects is used. As a result `delete[] ptr` calls the destructor of each of the wrapper class objects, in turn calling the `StringStore` destructor for their `d_stringStore` members. Example:

```
#include <iostream>
using namespace std;

class StringStore   // partially implemented
{
    public:
        ~StringStore();
};

inline StringStore::~StringStore()
{
    cout << "destructor called\n";
}

class Wrapper
{
    StringStore *d_stringStore;

    public:
```

```
        Wrapper();
        ~Wrapper();
};

inline Wrapper::Wrapper()
:
    d_stringStore(new StringStore())
{}
inline Wrapper::~Wrapper()
{
    delete d_stringStore;
}

int main()
{
    delete[] new StringStore *[4]; // memory leak: no destructor called
    cout << "===========\n";
    delete[] new Wrapper[4];       // OK: 4 x destructor called
}
/*
    Generated output:
===========
destructor called
destructor called
destructor called
destructor called
*/
```

8.2.2 The function set_new_handler()

The **C++** run-time system ensures that when memory allocation fails an error function is activated. By default this function throws a *bad_alloc* exception (see section 9.8), terminating the program. Therefore it is not necessary to check the return value of operator `new`. Operator `new`'s default behavior may be modified in various ways. One way to modify its behavior is to redefine the function that's called when memory allocation fails. Such a function must comply with the following requirements:

- it has no parameters;
- its return type is `void`.

A redefined error function might, e.g., print a message and terminate the program. The user-written error function becomes part of the allocation system through the function `set_new_handler`.

Such an error function is illustrated below[1]:

```
#include <iostream>
#include <string>
using namespace std;

void outOfMemory()
{
    cout << "Memory exhausted. Program terminates." << '\n';
    exit(1);
}
```

[1] This implementation applies to the Gnu **C/C++** requirements. Actually using the program given in the next example is not advised, as it will probably slow down your computer enormously due to the resulting use of the operating system's *swap area*.

```
int main()
{
    long allocated = 0;

    set_new_handler(outOfMemory);       // install error function

    while (true)                        // eat up all memory
    {
        memset(new int [100000], 0, 100000 * sizeof(int));
        allocated += 100000 * sizeof(int);
        cout << "Allocated " << allocated << " bytes\n";
    }
}
```

Once the new error function has been installed it is automatically invoked when memory allocation fails, and the program is terminated. Memory allocation may fail in indirectly called code as well, e.g., when constructing or using streams or when strings are duplicated by low-level functions.

So far for the theory. On some systems the 'out of memory' condition may actually never be reached, as the operating system may interfere before the run-time sypport system gets a chance to stop the program (see also `this link`[2]).

The standard **C** functions allocating memory (like `strdup`, `malloc`, `realloc` etc.) do not trigger the `new` handler when memory allocation fails and should be avoided in **C++** programs.

8.3 The assignment operator

In **C++** struct and class type objects can be directly assigned new values in the same way as this is possible in **C**. The default action of such an assignment for non-class type data members is a straight byte-by-byte copy from one data member to another. For now we'll use the following simple class `Person`:

```
class Person
{
    char *d_name;
    char *d_address;
    char *d_phone;

    public:
        Person();
        Person(char const *name, char const *addr, char const *phone);
        ~Person();
    private:
        char *strdupnew(char const *src);   // returns a copy of src.
};
```

`Person`'s data members are initialized to zeroes or to copies of the ASCII-Z strings passed to `Person`'s constructor, using some variant of `strdup`. Its destructor will return the allocated memory again.

Now consider the consequences of using `Person` objects in the following example:

```
void tmpPerson(Person const &person)
{
    Person tmp;
    tmp = person;
}
```

[2] http://www.linuxdevcenter.com/pub/a/linux/2006/11/30/linux-out-of-memory.html

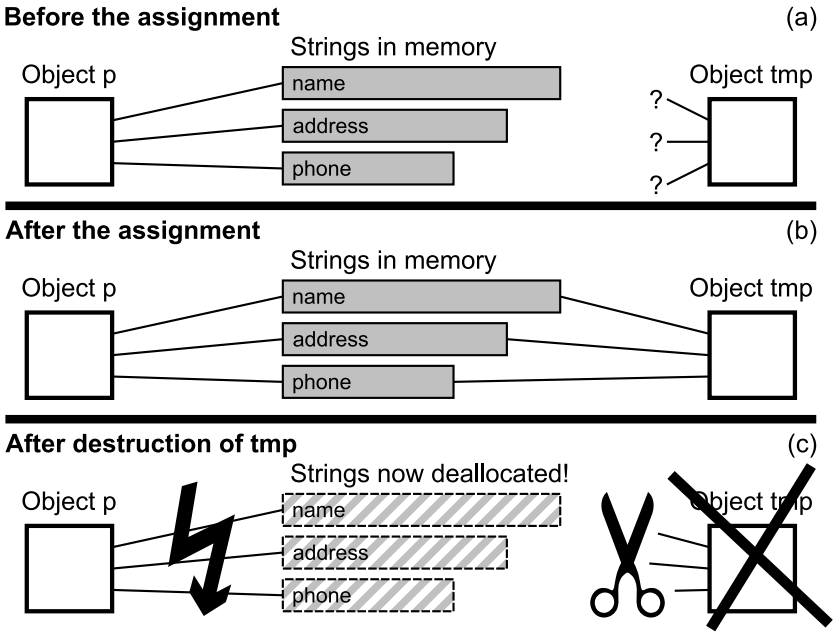

Figure 8.1: Private data and public interface functions of the class Person, using byte-by-byte assignment

Here's what happens when `tmpPerson` is called:

- it expects a reference to a `Person` as its parameter `person`.
- it defines a local object `tmp`, whose data members are initialized to zeroes.
- the object referenced by `person` is copied to `tmp`: `sizeof(Person)` number of bytes are copied from `person` to `tmp`.

Now a potentially dangerous situation has been created. The actual values in `person` are *pointers*, pointing to allocated memory. After the assignment this memory is addressed by two objects: `person` *and* `tmp`.

- The potentially dangerous situation develops into an acutely dangerous situation once the function `tmpPerson` terminates: `tmp` is destroyed. The destructor of the class `Person` releases the memory pointed to by the fields `d_name`, `d_address` and `d_phone`: unfortunately, this memory is also pointed at by `person`....

This problematic assignment is illustrated in Figure 8.1.

Having executed `tmpPerson`, the object referenced by `person` now contains pointers to deleted memory.

This is undoubtedly not a desired effect of using a function like `tmpPerson`. The deleted memory will likely be reused by subsequent allocations. The pointer members of `person` have effectively become *wild pointers*, as they don't point to allocated memory anymore. In general it can be concluded that

every class containing pointer data members is a potential candidate for trouble.

Fortunately, it is possible to prevent these troubles, as discussed next.

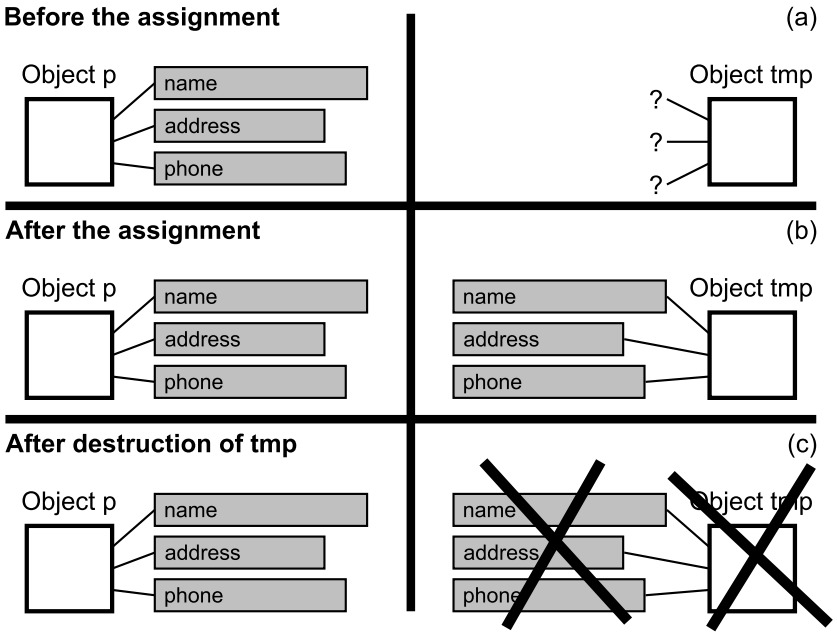

Figure 8.2: Private data and public interface functions of the class Person, using the 'correct' assignment.

8.3.1 Overloading the assignment operator

Obviously, the right way to assign one `Person` object to another, is **not** to copy the contents of the object bytewise. A better way is to make an equivalent object. One having its own allocated memory containing copies of the original strings.

The way to assign a `Person` object to another is illustrated in Figure 8.2. There are several ways to assign a `Person` object to another. One way would be to define a special member function to handle the assignment. The purpose of this member function would be to create a copy of an object having its own `name`, `address` and `phone` strings. Such a member function could be:

```
void Person::assign(Person const &other)
{
        // delete our own previously used memory
    delete[] d_name;
    delete[] d_address;
    delete[] d_phone;

        // copy the other Person's data
    d_name    = strdupnew(other.d_name);
    d_address = strdupnew(other.d_address);
    d_phone   = strdupnew(other.d_phone);
}
```

Using `assign` we could rewrite the offending function `tmpPerson`:

```
void tmpPerson(Person const &person)
{
    Person tmp;

        // tmp (having its own memory) holds a copy of person
    tmp.assign(person);

        // now it doesn't matter that tmp is destroyed..
```

```
}
```

This solution is valid, although it only solves a symptom solution. It requires the programmer to use a specific member function instead of the assignment operator. The original problem (assignment produces wild pointers) is still not solved. Since it is hard to 'strictly adhere to a rule' a way to solve the original problem is of course preferred.

Fortunately a solution exists using *operator overloading*: the possibility **C++** offers to redefine the actions of an operator in a given context. Operator overloading was briefly mentioned earlier, when the operators << and >> were redefined to be used with streams (like `cin`, `cout` and `cerr`), see section 3.1.4.

Overloading the assignment operator is probably the most common form of operator overloading in **C++**. A word of warning is appropriate, though. The fact that **C++** allows operator overloading does not mean that this feature should indiscriminately be used. Here's what you should keep in mind:

- operator overloading should be used in situations where an operator has a defined action, but this default action has undesired side effects in a given context. A clear example is the above assignment operator in the context of the class `Person`.
- operator overloading can be used in situations where the operator is commonly applied and no surprise is introduced when it's redefined. An example where operator overloading is appropriately used is found in the class `std::string`: assiging one string object to another provides the destination string with a copy of the contents of the source string. No surprises here.
- in all other cases a member function shold be defined instead of redefining an operator.

An operator should simply do what it is designed to do. The phrase that's often encountered in the context of operator overloading is *do as the `ints` do*. The way operators behave when applied to `ints` is what is expected, all other implementations probably cause surprises and confusion. Therefore, overloading the insertion (<<) and extraction (>>) operators in the context of streams is probably ill-chosen: the stream operations have nothing in common with bitwise shift operations.

8.3.1.1 The member 'operator=()'

To add operator overloading to a class, the class interface is simply provided with a (usually *public*) member function naming the particular operator. That member function is thereupon implemented.

To overload the assignment operator =, a member `operator=(Class const &rhs)` is added to the class interface. Note that the function name consists of two parts: the keyword `operator`, followed by the operator itself. When we augment a class interface with a member function `operator=`, then that operator is *redefined* for the class, which prevents the default operator from being used. In the previous section the function `assign` was provided to solve the problems resulting from using the default assignment operator. Rather than using an ordinary member function **C++** commonly uses a dedicated operator generalizing the operator's default behavior to the class in which it is defined.

The `assign` member mentioned before may be redefined as follows (the member `operator=` presented below is a first, rather unsophisticated, version of the overloaded assignment operator. It will shortly be improved):

```
class Person
{
    public:                             // extension of the class Person
                                        // earlier members are assumed.
        void operator=(Person const &other);
};
```

Its implementation could be

```
void Person::operator=(Person const &other)
{
    delete[] d_name;                    // delete old data
    delete[] d_address;
    delete[] d_phone;

    d_name = strdupnew(other.d_name);   // duplicate other's data
    d_address = strdupnew(other.d_address);
    d_phone = strdupnew(other.d_phone);
}
```

This member's actions are similar to those of the previously mentioned member `assign`, but this member is automatically called when the assignment operator = is used. Actually there are *two* ways to call overloaded operators as shown in the next example:

```
void tmpPerson(Person const &person)
{
    Person tmp;

    tmp = person;
    tmp.operator=(person);  // the same thing
}
```

Overloaded operators are seldom called explicitly, but an explicit call is required when the overloaded operator must be called from a pointer to an object:

```
void tmpPerson(Person const &person)
{
    Person *tmp = new Person;

    tmp->operator=(person);
    *tmp = person;          // yes, also possible...

    delete tmp;
}
```

8.4 The 'this' pointer

A member function of a given class is always called in combination with an object of its class. There is always an implicit 'substrate' for the function to act on. **C++** defines a keyword, `this`, to reach this substrate.

The `this` keyword is a pointer variable that always contains the address of the object for which the member function was called. The `this` pointer is implicitly declared by each member function (whether `public`, `protected`, or `private`). The `this` ponter is a constant pointer to an object of the member function's class. For example, the members of the class `Person` implicitly declare:

```
extern Person *const this;
```

A member function like `Person::name` could be implemented in two ways: with or without using the `this` pointer:

```
char const *Person::name() const    // implicitly using 'this'
{
    return d_name;
```

```
}

char const *Person::name() const    // explicitly using 'this'
{
    return this->d_name;
}
```

The `this` pointer is seldom explicitly used, but situations do exist where the `this` pointer is actually required (cf. chapter 16).

8.4.1 Sequential assignments and this

C++'s syntax allows for sequential assignments, with the assignment operator associating from right to left. In statements like:

```
a = b = c;
```

the expression `b = c` is evaluated first, and its result in turn is assigned to `a`.

The implementation of the overloaded assignment operator we've encountered thus far does not permit such constructions, as it returns `void`.

This imperfection can easily be remedied using the `this` pointer. The overloaded assignment operator expects a reference to an object of its class. It can also *return* a reference to an object of its class. This reference can then be used as an argument in sequential assignments.

The overloaded assignment operator commonly returns a reference to the current object (i.e., `*this`). The next version of the overloaded assignment operator for the class `Person` thus becomes:

```
Person &Person::operator=(Person const &other)
{
    delete[] d_address;
    delete[] d_name;
    delete[] d_phone;

    d_address = strdupnew(other.d_address);
    d_name = strdupnew(other.d_name);
    d_phone = strdupnew(other.d_phone);

    // return current object as a reference
    return *this;
}
```

Overloaded operators may themselves be overloaded. Consider the `string` class, having overloaded assignment operators `operator=(std::string const &rhs)`, `operator=(char const *rhs)`, and several more overloaded versions. These additional overloaded versions are there to handle different situations which are, as usual, recognized by their argument types. These overloaded versions all follow the same mold: when necessary dynamically allocated memory controlled by the object is deleted; new values are assigned using the overloaded operator's parameter values and `*this` is returned.

8.5 The copy constructor: initialization vs. assignment

Consider the class `StringStore`, introduced in section 8.2, once again. As it contains several primitive type data members as well as a pointer to dynamically allocated memory it needs a constructor, a

destructor, and an overloaded assignment operator. In fact the class offers two constructors: in addition to the default constructor it offers a a constructor expecting a `char const *const *` and a `size_t`.

Now consider the following code fragment. The statement references are discussed following the example:

```
int main(int argc, char **argv)
{
    StringStore s1(argv, argc);     // (1)
    StringStore s2;                 // (2)
    StringStore s3(s1);             // (3)

    s2 = s1;                        // (4)
}
```

- At 1 we see an initialization. The object `s1` is initialized using `main`'s parameters: `StringStore`'s second constructor is used.
- At 2 `StringStore`'s *default constructor* is used, initializing an empty `StringStore` object.
- At 3 yet another `StringStore` object is created, using a constructor accepting an existing `StringStore` object. This form of initializations has not yet been discussed. It is called a *copy construction* and the constructor performing the initialization is called the *copy constructor*. Copy constructions are also encountered in the following form:

  ```
  StringStore s3 = s1;
  ```

  This is a *construction* and therefore an *initialization*. It is not an *assignment* as an assignment needs a left-hand operand that has already been defined. **C++** allows the assignment syntax to be used for constructors having only one parameter. It is somewhat deprecated, though.
- At 4 we see a plain assignment.

In the above example three objects where defined, each using a different constructor. The actually used constructor was deduced from the constructor's argument list.

The copy constructor encountered here is new. It does not result in a compilation error even though it hasn't been declared in the class interface. This takes us to the following rule:

> A copy constructor is always available, even if it isn't declared in the class's interface.

The copy constructor made available by the compiler is also called the trivial copy constructor. Starting with the C++0x standard it can easily be suppressed (using the `= delete` idiom). The trivial copy constructor performs a byte-wise copy operation of the existing object's primitive data to the newly created object, calls copy constructors to intialize the object's class data members from their counterparts in the existing object and, when inheritance is used, calls the copy constructors of the base class(es) to initialize the new object's base classes.

Consequently, in the above example the trivial copy constructor is used. As it performs a byte-by-byte copy operation of the object's primitive type data members that is exactly what happens at stattement 3. By the time `s2` ceases to exist its destructor will delete its array of strings. Unfortunately `d_string` is of a primitive data type and so it also deletes `s1`'s data. Once again we encounter wild pointers as a result of an object going out of scope.

The remedy is easy: instead of using the trivial copy constructor a copy constructor must explicitly be added to the class's interface and its definition must prevent the wild pointers, comparably to the way this was realized in the overloaded assignment operator. An object's dynamically allocated memory is *duplicated*, so that it will contain its own allocated data. The copy constructor is simpler than the overloaded assignment operator in that it doesn't have to delete previously allocated memory. Since the object is going to be created no previously allocated memory already exists.

`StringStore`'s copy constructor can be implemented as follows:

```
StringStore::StringStore(StringStore const &other)
:
    d_string(new string[other.d_size]),
    d_size(other.d_size)
{
    for (size_t idx = 0; idx != d_size; ++idx)
        d_string[idx] = other.d_string[idx];
}
```

The copy constructor is always called when an object is initialized using another object of its class. Apart from the plain copy construction that we encountered thus far, here are other situations where the copy constructor is used:

- it is used when a function defines a class type value parameter rather than a pointer or a reference. The function's argument initializes the function's parameter using the copy constructor. Example:

```
void process(StringStore store) // no pointer, no reference
{
    store.at(3) = "modified";   // doesn't modify 'outer'
}

int main(int argc, char **argv)
{
    StringStore outer(argv, argc);
    process(outer);
}
```

- it is used when a function defines a class type value return type. Example:

```
StringStory copy(StringStore const &store)
{
    return store;
}
```

Here `store` is used to initialize `copy`'s return value. The returned `StringStore` object is a temporary, anonymous object that may be immediately used by code calling `copy` but no assumptions can be made about its lifetime thereafter.

8.5.1 Revising 'operator=()'

The overloaded assignment operator has characteristics also encountered with the copy constructor and the destructor:

- The *copying of (private) data* occurs (1) in the copy constructor and (2) in the overloaded assignment function.
- Allocated memory is deleted (1) in the overloaded assignment function and (2) in the destructor.

The copy constructor and the destructor clearly are required. If the overloaded assignment operator also needs to return allocated memory and to assign new values to its data members couldn't the destructor and copy constructor be used for that?

As we've seen in our discussion of the destructor (section 8.2) the destructor can explicitly be called, but that doesn't hold true for the (copy) constructor. But let's briefly summarize what an overloaded assignment operator is supposed to do:

- It should delete the dynamically allocated memory controlled by the current object;
- It should reassign the current object's data members using a provided existing object of its class.

The second part surely looks like a copy construction. Copy construction becomes even more attractive after realizing that the copy constructor also initializes any reference data members the class might have. Realizing the copy construction part is easy: just define a local object and initialize it using the assignment operator's const reference parameter, like this:

```
Strings &operator=(Strings const &other)
{
    Strings tmp(other);
    // more to follow
    return *this;
}
```

The optimization `operator=(String tmp)` is enticing, but let's postpone that for a little while (at least until section 8.6).

Now that we've done the copying part, what about the deleting part? And isn't there another slight problem as well? After all we copied all right, but not into our intended (current, `*this`) object.

At this point it's time to introduce *swapping*. Swapping two variables means that the two variables exchange their values. Many classes (e.g., `std::string`) offer `swap` members allowing us to swap two of their objects. The *Standard Template Library* (STL, cf. chapter 18) offers various functions related to swappping. There is even a `swap` *generic algorithm* (cf. section 19.1.61). That latter algorithm, however, begs the current question, as it is customarily implemented using the assignment operator, so it's somewhat problematic to use it when implementing the assignment operator.

As we've seen with the placement `new` operator objects can be constructed in blocks of memory of `sizeof(Class)` bytes large. And so, two objects of the same class each occupy `sizeof(Class)` bytes. To swap these objects we merely have to swap the contents of those `sizeof(Class)` bytes. This procedure may be applied to classes whose objects may be swapped using a member-by-member swapping operation and can also be used for classes having reference data members. Here is its implementation for a hypothetical class `Class`, resulting in very fast swapping:

```
#include <cstring>

void Class::swap(Class &other)
{
    char buffer[sizeof(Class)];
    memcpy(buffer, &other, sizeof(Class));
    memcpy(&other, this,   sizeof(Class));
    memcpy(this,   buffer, sizeof(Class));
}
```

Let's add `void swap(Strings &other)` to the class `Strings` and complete its `operator=` implementation:

```
Strings &operator=(Strings const &other)
{
    Strings tmp(other);
    swap(tmp);
    return *this;
}
```

This `operator=` implementation is generic: it can be applied to every class whose objects are directly swappable. How does it work?

- The information in the `other` object is used to initialize a local `tmp` object. This takes care of the copying part of the assignment operator.
- Calling `swap` ensures that the current object receives its new values.
- When `operator=` terminates its local `tmp` object ceases to exist and its destructor is called. But by now it contains the data previously owned by the current object, so *those* data are now returned. Which takes care of the destruction part of the assignment operation.

Nice?

8.6 The move constructor (C++0x)

Before the advent of the C++0x standard **C++** offered basically two ways to assign the information pointed to by a data member of a temporary object to an *lvalue* object. Either a copy constructor or reference counting had to be used. The C++0x standard adds *move semantics* to these two, allowing *transfer* of the data pointed to by a temporary object to its destination.

Our class `Strings` has, among other members a data member `string *d_string`. Clearly, `Strings` should define a copy constructor, a destructor and an overloaded assignment operator.

Now design a function `loadStrings(std::istream &in)` extracting the strings of a `Strings` object from `in`. As the `Strings` object doesn't exit yet, the `String` object filled by `loadStrings` is returned by value. The function `loadStrings` returns a temporary object, which is then used to initialize an external `Strings` object:

```
Strings loadStrings(std::istream &in)
{
    Strings ret;
    // load the strings into 'ret'
    return ret;
}
// usage:
Strings store(loadStrings(cin));
```

In this example two full copies of a `Strings` object are required:

- initializing `loadString`'s value return type from its local `Strings ret` object;
- initializing `store` from `loadString`'s return value

The *rvalue reference* concept allows us to improve this procedure. An *rvalue reference* binds to an anonymous temporary (r)value and the compiler is required to do so whenever possible. We, as programmers, must inform the compiler in what situations rvalue references can be handled. We do this by providing overloaded members defining rvalue reference parameters.

One such overloaded member is the *move constructor*. The move constructor is a constructor defining an rvalue reference to an object of its own class as parameter. Here is the declaration of the `Strings` class move constructor:

```
Strings(Strings &&tmp);
```

Move constructors are allowed to simply assign the values of pointer data members to their own pointer data members without requiring them to make a copy of the source's data first. Having done so the

temporary's pointer value is set to zero to prevent its destructor from destroying data now owned by the just constructed object. The move constructor has *grabbed* or *stolen* the data from the temporary object. This is OK as the temporary cannot be referred to again (as it is anonymous, it cannot be accessed by other code) and ceases to exist shortly after the constructor's call anyway. Here is the implementation of `Strings` move constructor:

```
Strings::Strings(Strings &&tmp)
:
    d_memory(tmp.d_memory),
    d_size(tmp.d_size),
    d_capacity(tmp.d_capacity)
{
    tmp.d_memory = 0;
}
```

Once a class becomes a *move-aware* class this awareness must extend to its destructor as well. With `Strings` this is not an issue as its destructor only executes `delete[] d_string`, but it becomes an issue in classes using, e.g., pointers to pointer data members. Their destructors must visit each of the array's pointers to delete the objects pointed to by the array's elements. Assuming that `Strings` `d_string` data member was defined as `string **d_string` the implementation of `String`'s move constructor may remain as-is, but the destructor must now inspect `d_string` to prevent the destruction loop from executing when it is zero:

```
Strings::~Strings()
{
    if (d_string == 0)
        return;
    for (string **end = d_string + d_size; end-- != d_string; )
        delete *end;
    delete[] d_string;
}
```

In addition to the move constructor other members defining `Class const &` parameters may also be overloaded with members expecting `Class &&` parameters. Here too the compiler will select these latter overloads if an anonymous temporary argument is provided. Let's consider the implications for a minute using the next example, assuming `Class` offers a move constructor and a copy constructor:

```
Class factory();

void fun(Class const &other);   // a
void fun(Class &&tmp);          // b

vold callee(Class &&tmp);
{
    Class object(factory());    // 1
    Class object2(object);      // 2
    fun(object);                // 3
    fun(factory());             // 4
    fun(tmp);                   // 5
}

int main()
{
    callee(factory());
}
```

- At 1 the move constructor is called. Object is initialized by an anonymous temporary `Class` object;

- At 2 the copy constructor is called. The constructor's argument is an existing local object;
- At 3 function a is called: `fun`'s argument is an existing local object;
- At 4 function b is called: `fun`'s argument is an anonymous temporary object;
- At 5 function a is called. At first sight this might be surprising, but `fun`'s argument is not an *anonymous* temporary object but a *named* temporary object. Overloads expecting rvalue references are *only* selected when passed *anonymous* temporary objects.

Realizing that `fun(tmp)` might be called twice the compiler's choice is understandable. If `tmp`'s data would have been grabbed at the first call, the second call would receive `tmp` without any data. But at the last call we might know that `tmp` is never used again and so we might like to ensure that `fun(Class &&)` is called. This can be realized by the following cast:

```
fun(reinterpret_cast<Class &&>(tmp)); // last call!
```

More often, though the shorthand `fun(std::move(tmp))` is used, already performing the required cast for us. `Std::move` is indirectly declared by many header files. If no header is already declaring `std::move` then include `utility`.

It is pointless to provide a function with an rvalue reference return type. The compiler decides whether or not to use an overloaded member expecting an rvalue reference on the basis of the provided argument: if it is an anonymous temporary it will call the overloaded member defining the rvalue refeerence parameter.

Classes not using pointer members pointing to memory controlled by its objects (and not having base classes doing so, see chapter 13) do not benefit from overloaded members expecting rvalue references.

The compiler, when selecting a function to call applies a fairly simple algorithm, and also considers copy elision. This is covered shortly (section 8.7).

8.6.1 Move-only classes (C++0x)

Classes may very well allow move semantics without offering copy semantics. Most stream classes belong to this category. Extending their definition with move semantics greatly enhances their usability. Once move semantics becomes available for such classes, so called *factory functions* (functions returning an object constructed by the function) can easily be implemented. E.g.,

```
// assume char *filename
ifstream inStream(openIstream(filename));
```

For this example to work an `ifstream` constructor must offer a move constructor. This way there will at any time be only one object referring to the open `istream`.

Once classes offer move semantics their objects can also safely be stored in standard containers. When such containers performs reallocation (e.g., when their sizes are enlarged) they will use the object's move constructors rather than their copy constructors. As move-only classes suppress copy semantics containers storing objects of move-only classes implement the correct behavior in that it is impossible to assign such containers to each other.

8.7 Copy Elision and Return Value Optimization

When the compiler selects a member function (or constructor) it will do so according to a simple set of rules, matching arguments with parameter types.

Below two tables are provided. The first table should be used in cases where a function argument has a name, the second table should be used in cases where the argument is anonymous. In each table select the const or non-const column and then use the topmost overloaded function that is available having the specified parameter type.

The tables do not handle functions defining value parameters. If a function has overloads expecting, respectively, a value parameter and some form of reference parameter the compiler reports an ambiguity when such a function is called. In the following selection procedure we may assume, without loss of generality, that this ambiguity does not occur and that all parameter types are reference parameters.

Parameter types matching a function's argument of type `T` if the argument is:

- a *named* argument (an lvalue or a named rvalue)

non-const	const
(T &)	
(T const &)	(T const &)

  Example: for an `int x` argument a function `fun(int &)` is selected rather than a function `fun(int const &)`. If no `fun(int &)` is available the `fun(int const &)` function is used. If neither is available the compiler reports an error.

- an *anonymous* argument (an anonymous temporary or a literal value)

non-const	const
(T &&)	
(T const &&)	(T const &&)
(T const &)	(T const &)

  Example: for an `int arg()` argument a function `fun(int &&)` is selected rather than a function `fun(int const &&)`. If both functions are unavailable but a `fun(int const &)` is available, that function is used. If none of these functions is available the compiler reports an error.

The tables show that eventually *all* arguments can be used with a function specifying a `T const &` parameter. For *anonymous* arguments a similar *catch all* is available having a higher priority: `T const &&` matches all anonymous arguments. Thus, if named and anonymous arguments are to be distinguished an `T const &&` overloaded function will catch all temporaries.

As we've seen the move constructor grabs the information from a temporary for its own use. That is OK as the temporary is going to be destroyed after that anyway. It also means that the temporary's data members are modified. This modification can safely be considered a *non-mutating operation* on the temporary. It may thus be modified even if it was passed to a function specifying a `T const &&` parameter. In cases like these consider using a `const_cast` to cast away the const-ness of the rvalue reference. The `Strings` move constructor encountered before might therefore also have been implemented as follows, handling both `Strings` and `Strings const` anonymous temporaries:

```
Strings::Strings(Strings const &&tmp)
:
    d_string(tmp.d_string),
    d_size(tmp.d_size)
{
    const_cast<Strings &>(tmp).d_string = 0;
}
```

Having defined appropriate copy and/or move constructors it may be somewhat surprising to learn that the compiler may decide to stay clear of a copy or move operation. After all making *no* copy and *not* moving is more efficient than copying or moving.

The option the compiler has to avoid making copies (or perform move operations) is called *copy elision* or *return value optimization*. In all situations where copy or move constructions are appropriate the

compiler may apply copy elision. Here are the rules. In sequence the compiler considers the following options, stopping once an option can be selected:

- if a copy or move constructor exists, try copy elision
- if a move constructor exists, move.
- if a copy constructor exists, copy.
- report an error

All modern compilers apply copy elision. Here are some examples where it may be encountered:

```
class Elide;

Elide fun()         // 1
{
    Elide ret;
    return ret;
}

void gun(Elide par);

Elide elide(fun()); // 2

gun(fun());         // 3
```

- At 1 `ret` may never exist. Instead of using `ret` and copying `ret` eventually to `fun`'s return value it may directly use the area used to contain `fun`'s return value.
- At 2 `fun`'s return value may never exist. Instead of defining an area containing `fun`'s return value and copying that return value to `elide` the compiler may decide to use `elide` to create `fun`'s return value in.
- At 3 the compiler may decide to do the same for gun's `par` parameter: `fun`'s return value is directly created in `par`'s area, thus eliding the copy operation from `fun`'s return value to `par`.

8.8 Plain Old Data (C++0x)

C++ inherited the struct concept from **C** and extended it with the class concept. Structs are still used in **C++**, mainly to store and pass around aggregates of different data types. A commonly term for these structs is *plain old data* (pod).

The standard pod concept in **C++** completely matches **C**'s struct concept. The C++0x standard, however, relaxes these requirements to some extent. In the C++0x standard pod is considerd to be a class or struct having the following characteristics:

- it has a trivial default constructor.
  If a type has some *trivial member* then the type (or its base class(es), cf. chapter 13) does not explicitly define that member. Rather, it is supplied by the compiler. A trivial default constructor leaves all its non-class data members unitialized and will call the default constructors of all its class data members. A class having a trivial default constructor does not define any constructor at all (nor does/do its base class/classes). It may also define the default constructor using the default constructor syntax introduced in section 7.4;
- it has a trivial copy constructor.
  A trivial copy constructor byte-wise copies the non-class data members from the provided existing class object and uses copy constructors to initialize its base class(es) and class data members with the information found in the provided existing class object;

- it has a trivial overloaded assignment operator.
  A trivial assignment operator performs a byte-wise copy of the non-class data members of the provided right-hand class object and uses overloaded assignment operators to assign new values to its class data members using the corresponding members of the provided right-hand class object;
- it has a trivial destructor.
  A trivial destructor calls the destructors of its base class(es) and class-type data members.

A *standard-layout* class or struct

- has only non-static data members that are themselves showing the standard-layout;
- has identical access control (public, private, protected) for all its non-static members;
- has no virtual functions (cf. chapter 14);
- has no virtual base classes (cf. chapter 14);
- has only base classes (cf. chapter 13) that are of standard-layout type;
- has no base classes of the same type as the first defined non-static member;
- has only one (in)direct base class having non-static members.

8.9 Conclusion

Four important extensions to classes were introduced in this chapter: the destructor, the copy constructor, the move constructor and the overloaded assignment operator. In addition the importance of *swapping*, especially in combination with the overloaded assignment operator, was stressed.

Classes having pointer data members, pointing to dynamically allocated memory controlled by the objects of those classes, are potential sources of memory leaks. The extensions introduced in this chapter implement the standard defense against such memory leaks.

Encapsulation (data hiding) allows us to ensure that the object's data integrity is maintained. The automatic activation of constructors and destructors greatly enhance our capabilities to ensure the data integrity of objects doing dynamic memory allocation.

A simple conclusion is therefore that classes whose objects allocate memory controlled by themselves must at least implement a *destructor*, an *overloaded assignment operator* and a *copy constructor*. Implementing a *move constructor* remains optional, but it allows us to use *factory functions* with classes *not* allowing copy construction and/or assignment.

In the end, assuming the availability of at least a copy or move constructor, the compiler might avoid them using *copy elision*. Copy elision is optional may be used by the compiler in all situations where otherwise a copy or move constructor would have been used.

Chapter 9

Exceptions

C supports several ways for a program to react to situations breaking the normal unhampered flow of a program:

- The function may notice the abnormality and issue a message. This is probably the least disastrous reaction a program may show.
- The function in which the abnormality is observed may decide to stop its intended task, returning an error code to its caller. This is a great example of postponing decisions: now the *calling function* is faced with a problem. Of course the calling function may act similarly, by passing the error code up to *its* caller.
- The function may decide that things are going out of hand, and may call `exit` to terminate the program completely. A tough way to handle a problem if only because the destructors of local objects aren't activated.
- The function may use a combination of the functions `setjmp` and `longjmp` to enforce non-local exits. This mechanism implements a kind of `goto` jump, allowing the program to continue at an outer level, skipping the intermediate levels which would have to be visited if a series of returns from nested functions would have been used.

In **C++** all these flow-breaking methods are still available. However, of the mentioned alternatives, `setjmp` and `longjmp` isn't frequently encountered in **C++** (or even in **C**) programs, due to the fact that the program flow is completely disrupted.

C++ offers *exceptions* as the preferred alternative to, e.g., `setjmp` and `longjmp`. Exceptions allow **C++** programs to perform a controlled non-local return, without the disadvantages of `longjmp` and `setjmp`.

Exceptions are the proper way to bail out of a situation which cannot be handled easily by a function itself, but which is not disastrous enough for a program to terminate completely. Also, exceptions provide a flexible layer of control between the short-range `return` and the crude `exit`.

In this chapter exceptions are covered. First an example will be given of the different impact exceptions and the `setjmp/longjmp` combination have on a programs. This example is followed by a discussion of the formal aspects of exceptions. In this part the guarantees our software should be able to offer when confronted with exceptions are presented. Exceptions and their guarantees have consequences for constructors and destructors. We'll encounter these consequences at the end of this chapter.

9.1 Exception syntax

Before contrasting the traditional **C** way of handling non-local gotos with exceptions let's introduce the syntactic elements that are involved when using exceptions.

- Exceptions are generated by a `throw` statement. The keyword `throw`, followed by an expression of a certain type, throws the expression value as an exception. In **C++** anything having value semantics may be thrown as an exception: an `int`, a `bool`, a `string`, etc. However, there also exists a *standard exception* type (cf. section 9.8) that may be used as *base class* (cf. chapter 13) when defining new exception types.

- Exceptions are generated within a well-defined local environment, called a `try`-block. The run-time support system ensures that all of the program's code is itself surrounded by a *global `try` block*. Thus, every exception generated by our code will always reach the boundary of at lease one `try`-block. A program terminates when an exception reaches the boundary of the global `try` block, and when this happens destructors of local and global objects that were alive at the point where the exception was generated are not called. This is not a desirable situation and therefore all exceptions should be generated within a `try`-block explicitly defined by the program. Here is an example of a string exception thrown from within a `try`-block:

```
try
{
    // any code can be defined here
    if (someConditionIsTrue)
        throw string("this is the std::string exception");
    // any code can be defined here
}
```

- `catch`: Immediately following the `try`-block, one or more `catch`-clauses must be defined. A `catch`-clause consists of a catch-header defining the type of the exception it can catch followed by a compound statement defining what to do with the caught exception:

```
catch (string const &msg)
{
    // statements in which the caught string object are handled
}
```

  Multiple `catch` clauses may appear underneath each other, one for each exception type that has to be caught. In general the `catch` clauses may appear in any order, but there are exceptions requiring a specific order. To avoid confusion it's best to put a `catch` clause for the most general exception last. At most *one* exception clause will be activated. **C++** does not support a **Java**-style `finally`-clause activated after completing a catch clause.

9.2 An example using exceptions

In the following example the same basic program is used. The program uses two classes, `Outer` and `Inner`.

First, an `Outer` object is defined in `main`, and its member `Outer::fun` is called. Then, in `Outer::fun` an `Inner` object is defined. Having defined the `Inner` object, its member `Inner::fun` is called.

That's about it. The function `Outer::fun` terminates calling `inner`'s destructor. Then the program terminates, activating `outer`'s destructor. Here is the basic program:

```
#include <iostream>
using namespace std;

class Inner
{
    public:
        Inner();
        ~Inner();
        void fun();
```

```
};
Inner::Inner()
{
    cout << "Inner constructor\n";
}
Inner::~Inner()
{
    cout << "Inner destructor\n";
}
void Inner::fun()
{
    cout << "Inner fun\n";
}

class Outer
{
    public:
        Outer();
        ~Outer();
        void fun();
};
Outer::Outer()
{
    cout << "Outer constructor\n";
}
Outer::~Outer()
{
    cout << "Outer destructor\n";
}
void Outer::fun()
{
    Inner in;
    cout << "Outer fun\n";
    in.fun();
}

int main()
{
    Outer out;
    out.fun();
}

/*
    Generated output:
Outer constructor
Inner constructor
Outer fun
Inner fun
Inner destructor
Outer destructor
*/
```

After compiling and running, the program's output is entirely as expected: the destructors are called in their correct order (reversing the calling sequence of the constructors).

Now let's focus our attention on two variants in which we simulate a non-fatal disastrous event in the `Inner::fun` function. This event must supposedly be handled near `main`'s end.

We'll consider two variants. In the first variant the event is handled by `setjmp` and `longjmp`; in the

second variant the event is handled using **C++**'s exception mechanism.

9.2.1 Anachronisms: 'setjmp' and 'longjmp'

The basic program from the previous section is slightly modified to contain a variable `jmp_buf jmpBuf` used by `setjmp` and `longjmp`.

The function `Inner::fun` calls `longjmp`, simulating a disastrous event, to be handled near `main`'s end. In `main` a target location for the long jump is defined through the function `setjmp`. `Setjmp`'s zero return indicates the initialization of the `jmp_buf` variable, in which case `Outer::fun` is called. This situation represents the 'normal flow'.

The program's return value is *only* zero if `Outer::fun` terminates normally. The program, however, is designed in such a way that this won't happen: `Inner::fun` calls `longjmp`. As a result the execution flow returns to the `setjmp` function. Now it will *not* return a zero return value. Consequently, after calling `Inner::fun` from `Outer::fun` `main`'s `if`-statement is entered and the program terminates with return value 1. Try to follow these steps when studying the following program source, which is a direct modification of the basic program given in section 9.2:

```
#include <iostream>
#include <setjmp.h>
#include <cstdlib>

using namespace std;

jmp_buf jmpBuf;

class Inner
{
    public:
        Inner();
        ~Inner();
        void fun();
};

Inner::Inner()
{
    cout << "Inner constructor\n";
}
void Inner::fun()
{
    cout << "Inner fun\n";
    longjmp(jmpBuf, 0);
}
Inner::~Inner()
{
    cout << "Inner destructor\n";
}

class Outer
{
    public:
        Outer();
        ~Outer();
        void fun();
};

Outer::Outer()
```

```
{
    cout << "Outer constructor\n";
}
Outer::~Outer()
{
    cout << "Outer destructor\n";
}
void Outer::fun()
{
    Inner in;
    cout << "Outer fun\n";
    in.fun();
}

int main()
{
    Outer out;

    if (setjmp(jmpBuf) != 0)
        return 1;

    out.fun();
}
/*
    Generated output:
Outer constructor
Inner constructor
Outer fun
Inner fun
Outer destructor
*/
```

This program's output clearly shows that `inner`'s destructor is not called. This is a direct consequence of the non-local jump performed by `longjmp`. Processing proceeds immediately from the `longjmp` call inside `Inner::fun` to `setjmp` in `main`. There, its return value is unequal zero, and the program terminates with return value 1. Because of the non-local jump `Inner::~Inner` is never executed: upon return to `main`'s `setjmp` the existing stack is simply broken down disregarding any destructors waiting to be called.

This example illustrates that the destructors of objects can easily be skipped when `longjmp` and `setjmp` are used and **C++** programs should therefore avoid those functions like the plague.

9.2.2 Exceptions: the preferred alternative

Exceptions are **C++**'s answer to the problems caused by `setjmp` and `longjmp`. Here is an example using exceptions. The program is once again derived from the basic program of section 9.2:

```
#include <iostream>
using namespace std;

class Inner
{
    public:
        Inner();
        ~Inner();
        void fun();
};
Inner::Inner()
```

```
{
    cout << "Inner constructor\n";
}
Inner::~Inner()
{
    cout << "Inner destructor\n";
}
void Inner::fun()
{
    cout << "Inner fun\n";
    throw 1;
    cout << "This statement is not executed\n";
}

class Outer
{
    public:
        Outer();
        ~Outer();
        void fun();
};

Outer::Outer()
{
    cout << "Outer constructor\n";
}
Outer::~Outer()
{
    cout << "Outer destructor\n";
}
void Outer::fun()
{
    Inner in;
    cout << "Outer fun\n";
    in.fun();
}

int main()
{
    Outer out;
    try
    {
        out.fun();
    }
    catch (int x)
    {}
}
/*
    Generated output:
Outer constructor
Inner constructor
Outer fun
Inner fun
Inner destructor
Outer destructor
*/
```

`Inner::fun` now throws an `int` exception where a `longjmp` was previously used. Since `in.fun` is called by `out.fun`, the exception is generated within the `try` block surrounding the `out.fun` call. As

an `int` value was thrown this value will reappear in the `catch` clause beyond the `try` block.

Now `Inner::fun` terminates by throwing an exception instead of calling `longjmp`. The exception is caught in `main`, and the program terminates. Now we see that `inner`'s destructor is properly called. It is interesting to note that `Inner::fun`'s execution really terminates at the `throw` statement: The `cout` statement, placed just beyond the `throw` statement, isn't executed.

What did this example teach us?

- Exceptions provide a means to break a function's (and program's) normal flow without having to use a cascade of `return`-statements, and without the need to terminate the program using blunt tools like **exit**(3).
- Exceptions do not disrupt the proper activation of destructors. Since `setjmp` and `longjmp` *do* distrupt the proper activation of destructors their use is strongly deprecated in **C++**.

9.3 Throwing exceptions

Exceptions are generated by `throw` statements. The `throw` keyword is followed by an expression, defining the thrown exception value. Example:

```
throw "Hello world";        // throws a char *
throw 18;                   // throws an int
throw string("hello");      // throws a string
```

Local objects cease to exist when a function terminates. This is no different for exceptions.

Objects defined locally in functions are automatically destroyed once exceptions thrown by these functions leave these functions. This also happens to objects thrown as exceptions. However, just before leaving the function context the object is copied and it is this copy that eventually reaches the approprate `catch` clause.

The following examples illustrates this process. `Object::fun` defines a local `Object toThrow`, that is thrown as an exception. The exception is caught in `main`. But by then the object originally thrown doesn't exist anymore, and `main` received a copy:

`Object`'s copy constructor is special in that it defines its name as `" (copy)"` to which the other object's name is appended. This allow us to monitor the construction and destruction of objects more closely. `Object::fun` generates an exception, and throws its locally defined object. Just before throwing the exception the program has produced the following output:

```
Constructor of 'main object'
Constructor of 'local object'
Calling fun of 'main object'
```

When the exception is generated the next line of output is produced:

```
Copy constructor for 'local object' (copy)
```

The local object is passed to `throw` where it is treated as a value argument, creating a copy of `toThrow`. This copy is thrown as the exception, and the local `toThrow` object ceases to exist. The thrown exception is now caught by the `catch` clause, defining an `Object` value parameter. Since this is a *value* parameter yet another copy is created. Thus, the program writes the following text:

```
Destructor of 'local object'
Copy constructor for 'local object' (copy) (copy)
```

The `catch` block now displays:

```
Caught exception
```

Following this `o`'s `hello` member is called, showing us that we indeed received a *copy of the copy* of the original `toThrow` object:

```
Hello by 'local object' (copy) (copy)
```

Then the program terminates and its remaining objects are now destroyed, reversing their order of creation:

```
Destructor of 'local object' (copy) (copy)
Destructor of 'local object' (copy)
Destructor of 'main object'
```

The copy created by the `catch` clause clearly is superficial. It can be avoided by defining object *reference parameters* in `catch` clauses: '`catch (Object &o)`'. The program now produces the following output:

```
Constructor of 'main object'
Constructor of 'local object'
Calling fun of 'main object'
Copy constructor for 'local object' (copy)
Destructor of 'local object'
Caught exception
Hello by 'local object' (copy)
Destructor of 'local object' (copy)
Destructor of 'main object'
```

Only a single copy of `toThrow` was created.

It's a bad idea to throw a *pointer* to a locally defined object. The pointer is thrown, but the object to which the pointer refers ceases to exist once the exception is thrown. The catcher receives a wild pointer. Bad news....

Let's summarize the above findings:

- Local objects are thrown as copied objects;
- Don't throw pointers to local objects;
- It is possible to throw pointers to *dynamically* generated objects. In this case one must take care that the generated object is properly deleted by the exception handler to prevent a memory leak.

Exceptions are thrown in situations where a function can't complete its assigned task, but the program is still able to continue. Imagine a program offering an interactive calculator. The program expects numeric expressions, which are evaluated. Expressions may show syntactic errors or it may be mathematically impossible to evaluate them. Maybe the calculator allows us to define and use variables and the user might refer to non-existing variables: plenty of reasons for the expression evaluation to fail, and so many reasons for exceptions to be thrown. None of those should terminate the program. Instead, the program's user is informed about the nature of the problem and is invited to enter another expression. Example:

```
if (!parse(expressionBuffer))           // parsing failed
    throw "Syntax error in expression";

if (!lookup(variableName))              // variable not found
```

```
        throw "Variable not defined";

    if (divisionByZero())                   // unable to do division
        throw "Division by zero is not defined";
```

Where these `throw` statements are located is irrelevant: they may be found deeply nested inside the program, or at a more superficial level. Furthermore, *functions* may be used to generate the exception to be thrown. An `Exception` object might support stream-like insertion operations allowing us to do, e.g.,

```
    if (!lookup(variableName))
        throw Exception() << "Undefined variable '" << variableName << "';
```

9.3.1 The empty 'throw' statement

Sometimes it is required to inspect a thrown exception. An exception catcher may decide to ignore the exception, to process the exception, to rethrow it after inspection or to change it into another kind of exception. For example, in a server-client application the client may submit requests to the server by entering them into a queue. Normally every request is eventually answered by the server. The server may reply the request was successfully processed, or that some sort of error has occurred. On the other hand, the server may have died, and the client should be able to discover this calamity, by not waiting indefinitely for the server to reply.

In this situation an intermediate exception handler is called for. A thrown exception is first inspected at the middle level. If possible it is processed there. If it is not possible to process the exception at the middle level, it is passed on, unaltered, to a more superficial level, where the really tough exceptions are handled.

By placing an *empty* `throw` statement in the exception handler's code the received exception is passed on to the next level that might be able to process that particular type of exception. The *retrown* exception is never handled by one of its neighboring exception handlers; it is always transferred to an exception handler at a more superficial level.

In our server-client situation a function

```
    initialExceptionHandler(string &exception)
```

could be designed to handle the `string` exception. The received message is inspected. If it's a simple message it's processed, otherwise the exception is passed on to an outer level. In `initialExceptionHandler`'s implementation the empty `throw` statement is used:

```
    void initialExceptionHandler(string &exception)
    {
        if (!plainMessage(exception))
            throw;

        handleTheMessage(exception);
    }
```

Below (section 9.5), the empty `throw` statement is used to pass on the exception received by a `catch`-block. Therefore, a function like `initialExceptionHandler` can be used for a variety of thrown exceptions, as long as their types match `initialExceptionHandler`'s parameter, which is a string.

The next example jumps slightly ahead, using some of the topics covered in chapter 14. The example may be skipped, though, without loss of continuity.

A basic exception handling class can be constructed from which specific exception types are derived. Suppose we have a class `Exception`, containing a member function `ExceptionType Exception::severity`.

This member function tells us (little wonder!) the severity of a thrown exception. It might be `Info`, `Notice`, `Warning`, `Error` or `Fatal`. The information contained in the exception depends on its severity and is processed by a function `handle`. In addition, all exceptions support a member function like `textMsg`, returning textual information about the exception in a `string`.

By defining a polymorphic function `handle` it can be made to behave differently, depending on the nature of a thrown exception, when called from a basic `Exception` pointer or reference.

In this case, a program may throw any of these five exception types. Let's assume that classes `Message` and `Warning` were derived from the class `Exception`, then the `handle` function matching the exception type will automatically be called by the following exception catcher:

```
catch(Exception &ex)
{
    cout << e.textMsg() << '\n';

    if
    (
        ex.severity() != ExceptionWarning
        &&
        ex.severity() != ExceptionMessage
    )
        throw;              // Pass on other types of Exceptions

    ex.handle();            // Process a message or a warning
}
```

Now anywhere in the `try` block preceding the exception handler `Exception` objects or objects of one of its derived classes may be thrown, that will all be caught by the above handler. E.g.,

```
throw Info();
throw Warning();
throw Notice();
throw Error();
throw Fatal();
```

9.4 The try block

The `try`-block surrounds `throw` statements. Remember that a program is always surrounded by a global `try` block, so `throw` statements may appear anywhere in your code. More often, though, `throw` statements are used in function bodies and such functions may be called from within `try` blocks.

A `try` block is defined by the keyword `try` followed by a compound statement. This block, in turn, *must* be followed by at least one `catch` handler:

```
try
{
            // any statements here
}
catch(...)  // at least one catch clause here
{}
```

`Try`-blocks are commonly nested, creating exception *levels*. For example, `main`'s code is surrounded by a `try`-block, forming an outer level handling exceptions. Within `main`'s `try`-block functions are called which may also contain `try`-blocks, forming the next exception level. As we have seen (section 9.3.1), exceptions thrown in inner level `try`-blocks may or may not be processed at that level. By placing an

empty `throw` statement in an exception handler, the thrown exception is passed on to the next (outer) level.

9.5 Catching exceptions

A `catch` clause consists of the keyword `catch` followed by a parameter list defining one parameter specifying type and (parameter) name of the exception the `catch` handler will catch. This name may then be used as a variable in the compound statement following the `catch` clause. Example:

```
catch (string &message)
{
    // code to handle the message
}
```

Primitive types and objects may be thrown as exceptions. It's a bad idea to throw a pointer or reference to a local object, but a pointer to a *dynamically* allocated object may be thrown as the exception handler will be able to delete the allocated memory thus preventing a memory leak. Throwing such a pointer is still dangerous as the exception handler will not be able to distinguish dynamically allocated memory and non-dynamically allocated memory, as illustrated by the next example:

```
try
{
    int x;
    int *xp = &x;

    if (condition1)
        throw &xp;

    xp = new int(0);
    if (condition2)
        throw xp;
}
catch (int *ptr)
{
    // delete ptr or not?
}
```

Close attention should be paid to the nature of the parameter of the exception handler, to make sure that when pointers to dynamically allocated memory are thrown the memory is returned once the handler has processed the pointer. In general pointers should not be thrown as exceptions. If dynamically allocated memory must be passed to a exception handler then the pointer should be wrapped in a smart pointer, like `unique_ptr` or `shared_ptr` (cf. sections 18.3 and 18.4).

Multiple `catch` handlers may follow a `try` block, each handler defining its own exception type. The *order* of the exception handlers is important. When an exception is thrown, the first exception handler matching the type of the thrown exception is used and remaining exception handlers are ignored. Eventually at most one exception handler following a `try`-block will be used. Normally this is of no concern as each exception has its own unique type.

Example: if exception handlers are defined for `char *`s and `void *`s then ASCII-Z strings will be caught by the former handler. Note that a `char *` can also be considered a `void *`, but the exception type matching procedure is smart enough to use the `char *` handler with the thrown ASCII-Z string. Handlers should be designed very type specific to catch the correspondingly typed exception. For example, `int`-exceptions are not caught by `double`-catchers, `char`-exceptions are not caught by `int`-catchers. Here is a little example illustrating that the order of the catchers is not important for types not having any hierarchal relationship to each other (i.e., `int` is not derived from `double`; `string` is not derived from ASCII-Z):

```
#include <iostream>
using namespace std;

int main()
{
    while (true)
    {
        try
        {
            string s;
            cout << "Enter a,c,i,s for ascii-z, char, int, string "
                                                    "exception\n";
            getline(cin, s);
            switch (s[0])
            {
                case 'a':
                    throw "ascii-z";
                case 'c':
                    throw 'c';
                case 'i':
                    throw 12;
                case 's':
                    throw string();
            }
        }
        catch (string const &)
        {
            cout << "string caught\n";
        }
        catch (char const *)
        {
            cout << "ASCII-Z string caught\n";
        }
        catch (double)
        {
            cout << "isn't caught at all\n";
        }
        catch (int)
        {
            cout << "int caught\n";
        }
        catch (char)
        {
            cout << "char caught\n";
        }
    }
}
```

Rather than defining specific exception handlers a specific class can be designed whose objects contain information about the exception. Such an approach was mentioned earlier, in section 9.3.1. Using this approach, there's only one handler required, since we *know* we don't throw other types of exceptions:

```
    try
    {
        // code throws only Exception pointers
    }
    catch (Exception &ex)
    {
        ex.handle();
```

```
}
```

When the code of an exception handler has been processed, execution continues beyond the last exception handler directly following the matching `try`-block (assuming the handler doesn't itself use flow control statements (like `return` or `throw`) to break the default flow of execution). The following cases can be distinguished:

- If *no* exception was thrown within the `try`-block no exception handler is activated, and execution continues from the last statement in the `try`-block to the first statement beyond the last `catch`-block.
- If an exception *was* thrown within the `try`-block but neither the current level nor an other level contains an appropriate exception handler, the program's default exception handler is called, aborting the program.
- If an exception was thrown from the `try`-block and an appropriate exception handler is available, then the code of that exception handler is executed. Following that, the program's execution continues at the first statement beyond the last `catch`-block.

All statements in a `try` block following an executed `throw`-statement are ignored. However, objects that were successfully constructed within the `try` block before executing the `throw` statement *are* called before any exception handler's code is executed.

9.5.1 The default catcher

At a certain level of the program only a limited set of handlers may actually be required. Exceptions whose types belong to that limited set are processed, all other exceptions are passed on to exception handlers of an outer level `try` block.

An intermediate type of exception handling may be implemented using the default exception handler, which must be (due to the hierarchal nature of exception catchers, discussed in section 9.5) placed beyond all other, more specific exception handlers.

This default exception handler cannot determine the actual type of the thrown exception and cannot determine the exception's value but it coould do some default processing. The exception is not lost, however, and the default exception handler may still use the empty `throw` statement (see section 9.3.1) to pass the exception on to an outer level. Here is an example showing this use of a default exception handler:

```
#include <iostream>
using namespace std;

int main()
{
    try
    {
        try
        {
            throw 12.25;    // no specific handler for doubles
        }
        catch (int value)
        {
            cout << "Inner level: caught int\n";
        }
        catch (...)
        {
            cout << "Inner level: generic handling of exceptions\n";
            throw;
```

```
            }
        }
        catch(double d)
        {
            cout << "Outer level may use the thrown double: " << d << '\n';
        }
    }
    /*
        Generated output:
    Inner level: generic handling of exceptions
    Outer level may use the thrown the double: 12.25
    */
```

The program's output illustrates that an empty `throw` statement in a default exception handler throws the received exception to the next (outer) level of exception catchers, keeping type and value of the thrown exception. Thus basic or generic exception handling can be accomplished at an inner level and specific handling, based on the type of the thrown expression, can then be provided at an outer level.

9.6 Declaring exception throwers

Functions defined elsewhere may be linked to code that uses these functions. Such functions are normally declared in header files, either as stand alone functions or as class member functions.

Those functions may of course throw exceptions. Declarations of such functions may contain a *function throw list* or *exception specification list* specifying the types of the exceptions that can be thrown by the function. For example, a function that may throw '`char *`' and '`int`' exceptions can be declared as

```
void exceptionThrower() throw(char *, int);
```

A function throw list immediately follows the function header (and it also follows a possible `const` specifier). Throw lists may be empty. It has the following general form:

```
throw([type1 [, type2, type3, ...]])
```

If a function is guaranteed not to throw exceptions an empty function throw list may be used. E.g.,

```
void noExceptions() throw ();
```

In all cases, the function header used in the function definition must exactly match the function header used in the declaration, including a possibly empty function throw list.

A function for which a function throw list is specified may only throw exceptions of the types mentioned in its throw list. A *run-time error* occurs if it does. other types of exceptions than those mentioned in the function throw list. Example: the function `charPintThrower` shown below clearly throws a `char const *` exception. Since `intThrower` may throw an `int` exception, the function throw list of `charPintThrower` must *also* contain `int`.

```
#include <iostream>
using namespace std;

void charPintThrower() throw(char const *, int);

class Thrower
{
    public:
```

```
        void intThrower(int) const throw(int);
};

void Thrower::intThrower(int x) const throw(int)
{
    if (x)
        throw x;
}

void charPintThrower() throw(char const *, int)
{
    int x;

    cerr << "Enter an int: ";
    cin >> x;

    Thrower().intThrower(x);
    throw "this text is thrown if 0 was entered";
}

void runTimeError() throw(int)
{
    throw 12.5;
}

int main()
{
    try
    {
         charPintThrower();
    }
    catch (char const *message)
    {
        cerr << "Text exception: " << message << endl;
    }
    catch (int value)
    {
        cerr << "Int exception: " << value << endl;
    }
    try
    {
        cerr << "Generating a run-time error\n";
        runTimeError();
    }
    catch(...)
    {
        cerr << "not reached\n";
    }
}
```

A function without a throw list may throw any kind of exception. Without a function throw list the program's designer is responsible for providing the correct handlers.

For various reason declaring exception throwers is a questionable activity. Declaring exception throwers does not mean that the compiler will check whether an improper exception is thrown. Rather, the function will be surrounded by additional code in which the actual exception that is thrown is processed. Instead of compile time checks one gets run-time overhead, resulting in additional code (and execution time) thay is added to the function's code. One could write, e.g.,

```
void fun() throw (int)
{
    // code of this function, throwing exceptions
}
```

but the function would be compiled to something like the following (cf. section 9.10 for the use of `try` immediately following the function's header and section 9.8 for a description of `bad_exception`):

```
void fun()
try         // this code resulting from throw(int)
{
            // the function's code, throwing all kinds of exceptions
}
catch (int) // remaining code resulting from throw(int)
{
    throw;  // rethrow the exception, so it can be caught by the
            // 'intended' handler
}
catch (...) // catch any other exception
{
    throw bad_exception;
}
```

Run-time overhead is caused by doubling the number of thrown and caught exceptions. Without a throw list a thrown `int` is simply caught by its intended handler; with a throw list the `int` is *first* caught by the 'safeguarding' handler added to the function. In there it is *rethrown* to be caught by its intended handler next.

9.7 Iostreams and exceptions

The **C++** I/O library was used well before exceptions were available in **C++**. Hence, normally the classes of the iostream library do not throw exceptions. However, it is possible to modify that behavior using the `ios::exceptions` member function. This function has two overloaded versions:

- `iostate exceptions()`:

  this member returns the state flags for which the stream will throw exceptions;

- `void exceptions(iostate state)`

  this member causes the stream to throw an exception when state `state` is observed.

In the I/O library, exceptions are objects of the class `ios::failure`, derived from `ios::exception`. A `std::string const &message` may be specified when defining a `failure` object. Its message may then be retrieved using its `virtual char const *what() const` member.

Exceptions should be used in exceptional circumstances. Therefore, we think it is questionable to have stream objects throw exceptions for fairly normal situations like `EOF`. Using exceptions to handle input errors might be defensible (e.g., in situations where input errors should not occur and imply a corrupted file) but often aborting the program with an appropriate error message would probably be the more appropriate action. As an example consider the following interactive program using exceptions to catch incorrect input:

```
#include <iostream>
#include <climits>
using namespace::std;
```

```
int main()
{
    cin.exceptions(ios::failbit);   // throw exception on fail
    while (true)
    {
        try
        {
            cout << "enter a number: ";
            int value;
            cin >> value;
            cout << "you entered " << value << endl;
        }
        catch (ios::failure const &problem)
        {
            cout << problem.what() << endl;
            cin.clear();
            cin.ignore(INT_MAX, '\n');  // ignore the faulty line
        }
    }
}
```

9.8 Standard Exceptions

All data types may be thrown as exceptions. Several exception types are defined by the **C++** standard. All of these *standard exceptions* are class types by themselves, but also offer all facilities of the `std::exception` class and objects of the standard exception classes may also be considered objects of the `std::exception` class. The `std::exception` class offers the member

```
char const *what() const;
```

describing in a short textual message the nature of the exception.

C++ defines the following standard exception classes:

- `std::bad_alloc`: thrown when `operator new` fails;
- `std::bad_exception`: thrown when a function tries to generate another type of exception than declared in its function throw list;
- `std::bad_cast`: thrown in the context of *polymorphism* (see section 14.5.1);
- `std::bad_typeid`: also thrown in the context of *polymorphism* (see section 14.5.2);

9.9 Exception guarantees

Software should be *exception safe*: the program should continue to work according to its specifications in the face of exceptions. It is not always easy to realize exception safety. In this section some guidelines and terminology is introduced when discussing exception safety.

Since exceptions may be generated from within all **C++** functions exceptions may be generated in many situations, not all of which will be immediately and intuitively clear. Consider the following function and ask yourself at which points exceptions may be thrown:

```
void fun()
```

```
{
    X x;
    cout << x;
    X *xp = new X(x);
    cout << (x + *xp);
    delete xp;
}
```

If it can be assumed that `cout` as used above does not throw an exception there are at least 13 opportunities for exceptions to be thrown:

- `X x`: the default constructor could throw an exception (#1)
- `cout << x`: the overloaded insertion operator could throw an exception (#2), but its rvalue argument might not be an `X` but, e.g., an `int`, and so `X::operator int() const` could be called which offers yet another opportunity for an exception (#3).
- `*xp = new X(x)`: the copy constructor may throw an exception (#4) and operator new (#5a) too. But did you realize that this latter exception might not be thrown from `::new`, but from, e.g., `X`'s own overload of `operator new`? (#5b)
- `cout << (x + *xp)`: we might be seduced into thinking that two `X` objects are added it doesn't need to be that way. A separate class `Y` might exist and `X` may have a conversion operator `operator Y() const`, and `operator+(Y const &lhs, X const &rhs)`, `operator+(X const &lhs, Y const &rhs)`, and `operator+(X const &lhs, X const &rhs)` might all exist. So, if the conversion operator exists, then depending on the kind of overload of `operator+` that is defined either the addition's left-hand side operand (#6), right-hand side operand (#7), or `operator+` itself (#8) may throw an exception. The resulting value may again be of any type and so the overloaded `cout << return-type-of-operator+` operator may throw an exception (#9). Since `operator+` returns a temporary object it is destroyed shortly after its use. `X`'s destructor *could* throw an exception (#10).
- `delete xp`: whenever `operator new` is overloaded `operator delete` should be overloaded as well and may throw an exception (#11). And of course, `X`'s destructor might again throw an exception (#12).
- `}`: when the function terminates the local `x` object is destroyed: again an exception could be thrown (#13).

It is stressed here (and further discussed in section 9.11) that although it is possible for exceptions to leave destructors this would violate the **C++** standard and so it must be prevented in well-behaving **C++** programs.

How can we expect to create working programs when exceptions might be thrown at this many situations?

Exceptions may be generated in a great many situations, but serious problems will be prevented when we're able to provide at least one of the following exception guarantees:

- The *basic guarantee*: no resources are leaked. In practice this means: all allocated memory is properly returned when exceptions are thrown.
- The *strong guarantee*: the program's state remains unaltered when an exception is thrown (as an example: the canonical form of the overloaded assignment operator provides this guarantee)
- The *nothrow* guarantee: this applies to code for which it can be proven that no exception will be thrown from it.

9.9.1 The basic guarantee

The *basic guarantee* dicates that functions that failing to complete their assigned tasks must return all allocated resources, usually memory, before terminating. Since practically all functions and operators may throw exceptions and since a function may repeatedly allocate resources the blueprint of a function allocating resources shown below defines a try block to catch all exceptions that might be thrown. The catch handler's task is to return all allocated resources and then retrow the exception.

```
void allocator(X **xDest, Y **yDest)
{
    X *xp = 0;              // non-throwing preamble
    Y *yp = 0;

    try                     // this part might throw
    {
        xp = new X[nX];     // alternatively: allocate one object
        yp = new Y[nY];
    }
    catch(...)
    {
        delete xp;
        throw;
    }

    delete[] *xDest;        // non-throwing postamble
    *xDest = xp;
    delete[] *yDest;
    *yDest = yp;
}
```

In the pre-try code the pointers to receive the addresses returned by the operator `new` calls are initialized to 0. Since the catch handler must be able to return allocated memory they must be available outside of the `try` block. If the allocation succeeds the memory pointed to by the destination pointers is returned and then the pointers are given new values.

Allocation and or initialization might fail. If allocation fails `new` throws a `std::bad_alloc` exception and the catch handler simply deletes 0 pointers which is OK.

If allocation succeeds but the construction of (some) of the objects fails by throwing an exception then the following is *guaranteed* to happen:

- The destructors of all successfully allocated objects are called;
- The dynamically allocated memory to contain the objects is returned

Consequently, no memory will leak when `new` fails. Inside the above `try` block `new X` may fail: this will keep the 0-pointers intact and so the catch handler merely deletes 0 pointers. When `new Y` fails `xp` will point to allocated memory and so it must be returned. This happens inside the catch handler. The final pointer (here: `yp`) will only be unequal zero when `new Y` properly completes, so there's no need for the catch handler to return the memory pointed at by `yp`.

9.9.2 The strong guarantee

The *strong guarantee* dicates that an object's state should not change in the face of exceptions. This is realized by performing all operations that might throw on a separate copy of the data. If all this succeeds then the current object and its (now successfully modified) copy are swapped. An example of this approach can be observed in the canonical overloaded assignment operator:

```
Class &operator=(Class const &other)
{
    Class tmp(other);
    swap(other);
    return *this;
}
```

The copy construction might throw an exception, but this keeps the current object's state intact. If the copy construction succeeds `swap` swaps the current object's contents with `tmp`'s contents and returns a reference to the current object. For this to succeed it must be guaranteed that `swap` won't throw an exception. Returning a reference (or a value of a primitive data type) is also guaranteed not to throw exceptions. The canonical form of the overloaded assignment operator therefore meets the requirements of the strong guarantee.

Some rules of thumb were formulated that relate to the strong guarantee (cf. Sutter, H., *Exceptional C++*, Addison-Wesley, 2000). E.g.,

- All the code that might throw an exception affecting the current state of an object should perform its tasks separately from the data controlled by the object. Once this code has performed its tasks without throwing an exception replace the object's data by the new data.
- Member functions modifying their object's data should not return original (contained) objects by value.

The canonical assignment operator is a good example of the first rule of thumb. Another example is found in classes storing objects. Consider a class `PersonDb` storing multiple `Person` objects. Such a class might offer a member `void add(Person const &next)`. A plain implementation of this function (merely intended to show the application of the first rule of thumb, but otherwise completely disregarding efficiency considerations) might be:

```
void PersonDb::newAppend(Person const &next)
{
    Person *tmp = 0;
    try
    {
        tmp = new Person[d_size + 1];
        for (size_t idx = 0; idx < d_size; ++idx)
            tmp[idx] = d_data[idx];
        tmp[d_size] = next;
    }
    catch (...)
    {
        delete[] tmp;
        throw;
    }
    return tmp;
}

void PersonDb::add(Person const &next)
{
    Person *tmp = newAppend(next);
    delete[] d_data;
    d_data = tmp;
    ++d_size;
}
```

The (private) `newAppend` member's task is to create a copy of the currently allocated `Person` objects, including the data of the next `Person` object. Its `catch` handler catches any exception that might be

thrown during the allocation or copy process and will return all memory allocated so far, rethrowing the exception. The function is *exception neutral* as it propagates all its exceptions to its caller. The function also doesn't modify the `PersonDb` object's data, so it meets the strong exception guarantee. Returning from `newAppend` `addPerson` may now modify its data. Its existing data are returned and its `d_data` pointer is made to point to the newly created array of `Person` objects. Finally its `d_size` is incremented. As these three steps don't throw exceptions `add` too meets the strong guarantee.

The second rule of thumb (member functions modifying their object's data should not return original (contained) objects by value) may be illustrated using a member `PersonDb::erase(size_t idx)`. Here is an implementation attempting to return the original `d_data[idx]` object:

```
Person PersonData::erase(size_t idx)
{
    if (idx >= d_size)
        throw string("Array bounds exceeded");
    Person ret(d_data[idx]);
    Person *tmp = copyAllBut(idx);
    delete[] d_data;
    d_data = tmp;
    --d_size;
    return ret;
}
```

Although copy elision will usually prevent the copy constructor from being used when returning `ret`, this is not guaranteed to happen and a copy constructor *may* throw an exception. If that happens the function has irrevocably mutated the `PersonDb`'s data, thus losing the strong guarantee.

Rather than returning `d_data[idx]` by value it might be assigned to an external `Person` object befor mutating `PersonDb`'s data:

```
void PersonData::erase(Person *dest, size_t idx)
{
    if (idx >= d_size)
        throw string("Array bounds exceeded");
    *dest = d_data[idx];
    Person *tmp = copyAllBut(idx);
    delete[] d_data;
    d_data = tmp;
    --d_size;
}
```

This modification works, but changes the original assignment of creating a member returning the original object. However, both functions suffer from a task overload as they modify `PersonDb`'s data and also return an original object. In situations like these the *one-function-one-responsibility* rule of thumb should be kept in mind: a function should have a single, well defined responsibility.

The preferred approach therefore is to retrieve `PersonDb`'s objects using a member like `Person const &at(size_t idx) const` and to erase an object using a separate member like `void PersonData::erase(size_t idx)`.

9.9.3 The nothrow guarantee

Exception safety can only be realized if some functions and operations are guaranteed *not* to throw exceptions. This is called the *nothrow guarantee*. An example of a function that must offer the nothrow guarantee is the `swap` function. Consider once again the canonical overloaded assignment operator:

```
Class &operator=(Class const &other)
```

```
    {
        Class tmp(other);
        swap(other);
        return *this;
    }
```

If `swap` would be allowed to throw exceptions then it would most likely leave the current object in a partially swapped state. As a result the current object's state would most likely have been changed. As `tmp` will have been destroyed by the time a catch handler receives the thrown exception it will be very difficult (as in: impossible) to retrieve the object's original state. Losing the strong guarantee as a consequence.

The `swap` function must therefore offer the nothrow guarantee. It must have been designed as if using the following prototype (don't do this: see also section 9.6):

```
    void Class::swap(Class &other) throw();
```

Likewise, `operator delete` and `operator delete[]` offer the nothrow guarantee, and according to the **C++** standard destructors may themselves not throw exceptions (if they do their behavior is formally undefined, see also section 9.11 below).

Since the **C** programming language does not define the exception concept functions from the standard **C** library offer the nothrow guarantee by implication. This allowed us to define the generic `swap` function in section 8.5.1 using **memcpy**(3).

Operations on primitive types offer the nothrow guarantee. Pointers may be reassigned, references may be returned etc. etc. without having to worry about exceptions that might be thrown.

9.10 Function try blocks

Exceptions may be generated while a constructor is initializing its members. How can exceptions generated in such situations be caught by the constructor itself, rather than outside the constructor? The intuitive solution, nesting the object construction in a `try` block does not solve the problem. The exception by then has left the constructor and the object we intended to construct isn't visible anymore.

Using a nested `try` block is illustrated in the next example, where `process` defines an object of class `PersonDb`. Assuming that `PersonDb`'s constructor throws an exception, there is no way we can access the resources that might have been allocated by `PersonDb`'s constructor from the catch handler as the `db` object is out of scope:

```
int main(int argc, char **argv)
{
    try
    {
        PersonDb pdb(argc, argv);   // may throw exceptions
        ...                         // main()'s other code
    }
    catch(...)                      // and/or other handlers
    {
        ...                         // pdb is inaccessible from here
    }
}
```

Although all objects and variables defined inside a `try` block are inaccessible from its associated catch handlers, object data members were available before starting the `try` block and so they may be accessed from a catch handler. In the following example the catch handler in `PersonDb`'s constructor is able to access its `d_data` member:

```
PersonDb::PersonDb(int argc, char **argv)
:
    d_data(0),
    d_size(0)
{
    try
    {
        initialize(argc, argv);
    }
    catch(...)
    {
        // d_data, d_size: accessible
    }
}
```

Unfortunately, this does not help us much. The `initialize` member will not be able to reassign `d_data` and `d_size` if `PersonDb const pdb` was defined; the `initialize` member should at least offer the basic exception guarantee and return any resources it has acquired before terminating due to a trown exception; and although `d_data` and `d_size` offer the nothrow guarantee as they are of primitive data types a class type data member might throw an exception, possibly resulting in violation of the basic guarantee.

In the next implementation of `PersonDb` assume that constructor receives a pointer to an already allocated block of `Person` objects. It will own the data and is responsible for its eventual destruction. Moreover, `d_data` and `d_size` are also used by a composed object `PersonDbSupport`, having a constructor expecting a `Person const *` and `size_t` argument. Our next implementation may then look something like this:

```
PersonDb::PersonDb(Person *pData, size_t size)
:
    d_data(pData),
    d_size(size),
    d_support(d_data, d_size)
{
    // no further actions
}
```

This setup allows us to define a `PersonDb const &pdb`. Unfortunately, `PersonDb` cannot offer the basic guarantee. If `PersonDbSupport`'s constructor throws an exception it isn't caught although `d_data` already points to allocated memory.

The *function try block* offers a solution for this problem. A function try block consists of a `try` block and its associated handlers. The function `try` block starts *immediately* after the function header, and its block defines the function body. With constructors base class and data member initializers may be placed between the `try` keyword and the opening curly brace. Here is our final implementation of `PersonDb`, now offering the basic guarantee:

```
PersonDb::PersonDb(Person *pData, size_t size)
try
:
    d_data(pData),
    d_size(size),
    d_support(d_data, d_size)
{}
catch (...)
{
    delete[] d_data;
}
```

Let's have a look at a stripped-down example. A constructor defines a function try block. The exception thrown by the `Throw` object is initially caught by the object itself. Then it is rethrown. The surrounding `Composer`'s constructor also defines a function try block, `Throw`'s rethrown exception is properly caught by `Composer`'s exception handler, even though the exception was generated from within its member initializer list:

```
#include <iostream>

class Throw
{
    public:
        Throw(int value)
        try
        {
            throw value;
        }
        catch(...)
        {
            std::cout << "Throw's exception handled locally by Throw()\n";
            throw;
        }
};

class Composer
{
    Throw d_t;
    public:
        Composer()
        try             // NOTE: try precedes initializer list
        :
            d_t(5)
        {}
        catch(...)
        {
            std::cout << "Composer() caught exception as well\n";
        }
};

int main()
{
    Composer c;
}
```

When running this example, we're in for a nasty surprise: the program runs and then breaks with an *abort exception*. Here is the output it produces, the last two lines being added by the system's final catch-all handler, catching all remaining uncaught exceptions:

```
Throw's exception handled locally by Throw()
Composer() caught exception as well
terminate called after throwing an instance of 'int'
Abort
```

The reason for this is documented in the **C++** standard: at the end of a catch-handler belonging to a constructor or destructor function try block, the original exception is automatically rethrown.

The exception is not rethrown if the handler itself throws another exception, offering the constructor or destructor a way to replace a thrown exception by another one. The exception is only rethrown if it reaches the end of the catch handler of a constructor or destructor function try block. Exceptions caught by nested catch handlers are not automatically rethrown.

As only constructors and destructors rethrow exceptions caught by their function try block catch handlers the run-time error encountered in the above example may simply be repaired by providing `main` with its own function try block:

```
int main()
try
{
    Composer c;
}
catch (...)
{}
```

Now the program runs as planned, producing the following output:

```
Throw's exception handled locally by Throw()
Composer() caught exception as well
```

A final note: if a function defining a function try block also declares an exception throw list then only the types of rethrown exceptions must match the types mentioned in the throw list.

9.11 Exceptions in constructors and destructors

Only completely constructed objects are automatically destroyed by the **C++** run-time system. Although this may sound like a truism, there is a subtlety here. If the construction of an object fails for some reason, the object's destructor will not be called once the object goes out of scope. This could happen if an exception generated by the constructor is not caught by the constructor. If the exception is thrown *after* the object has allocated some memory, then that memory will not be returned by the object's destructor as the destructor isn't called because the object hasn't completely been constructed.

The following example illustrates this situation in its prototypical form. The constructor of the class `Incomplete` first displays a message and then throws an exception. Its destructor also displays a message:

```
class Incomplete
{
    public:
        Incomplete()
        {
            cerr << "Allocated some memory\n";
            throw 0;
        }
        ~Incomplete()
        {
            cerr << "Destroying the allocated memory\n";
        }
};
```

Next, `main()` creates an `Incomplete` object inside a `try` block. Any exception that may be generated is subsequently caught:

```
int main()
{
    try
    {
        cerr << "Creating 'Incomplete' object\n";
```

```
            Incomplete();
            cerr << "Object constructed\n";
        }
        catch(...)
        {
            cerr << "Caught exception\n";
        }
    }
```

When this program is run, it produces the following output:

```
Creating 'Incomplete' object
Allocated some memory
Caught exception
```

Thus, if `Incomplete`'s constructor would actually have allocated some memory, the program would suffer from a memory leak. To prevent this from happening, the following counter measures are available:

- Prevent the exceptions from leaving the constructor.
  If part of the constructor's body may generate exceptions, then this part may be surrounded by a `try` block, allowing the exception to be caught by the constructor itself. This is approach is defensible when the constructor is able to repair the cause of the exception and to complete its construction as a valid object.
- If an exception is generated by a base class constructor or by a member initializing constructor then a `try` block within the constructor's body won't be able to catch the thrown exception. This *always* results in the exception leaving the constructor and the constructor will never be considered properly constructed. A `try` block may include the member initializers, and the `try` block's compound statement becomes the constructor's body as in the following example:

  ```
  class Incomplete2
  {
      Composed d_composed;
      public:
          Incomplete2()
          try
          :
              d_composed(/* arguments */)
          {
              // body
          }
          catch (...)
          {}
  };
  ```

  An exception thrown by either the member initializers or the body will result in the execution never reaching the body's closing curly brace. Instead the catch clause is reached. Since the constructor's body isn't properly completed the object is not considered properly constructed and eventually the object's destructor won't be called.

The catch clause of a constructor's function `try` block behaves slightly different than a catch clause of an ordinary function `try` block. An exception reaching a constructor's function `try` block may be transformed into another exception (which is thrown from the catch clause) but if no exception is explicitly thrown from the catch clause the exception originally reaching the catch clause is always rethrown. Consequently, there's no way to confine an exception thrown from a base class constructor or from a member initializer to the constructor: such an exception will *always* propagate to a more shallow block and the object's construction will always be considered incomplete.

Consequently, if incompletely constructed objects throw exceptions then the constructor's catch clause is responsible for preventing memory (generally: resource) leaks. There are several ways to realize this:

- When multiple inheritance is used: if initial base classes have properly been constructed and a later base class throws, then the initial base class destructors will automatically be destroyed (as they are themselves fully constructed objects)
- When composition is used: already constructed composed objects will automatically be destroyed (as they are fully constructed objects)
- Instead of using plain pointers *smart pointers* (cf. section 18.4) should be used to manage dynamically allocated memory. In this case, if the constructor throws either before or after the allocation of the dynamic memory allocated memory will properly be returned as `shared_ptr` objects are objects.
- If plain pointer *must* be used then the constructor's body should use local pointers when dynamically allocating memory and assign the local pointers to the plain pointer data members in a final *nothrow* section. For example:

```
class Incomplete2
{
    Composed d_composed;
    char *d_cp;
    int *d_ip;

    public:
        Incomplete2(size_t nChars, size_t nInts)
        try
        :
            d_composed(/* arguments */)     // may throw
            d_cp(0),
            d_ip(0)
        {
            preamble();                     // may throw
            try
            {
                d_cp = new char[nChars];    // may throw
                d_ip = new int[nChars];     // may throw
                postamble();                // may throw
            }
            catch(...)
            {
                delete[] d_cp;              // clean up
                delete[] d_ip;

                throw;
            }
            // onlly nohrow operations here
        }
        catch (...)
        {}
};
```

According to the **C++** standard exceptions thrown by destructors may not leave their bodies. Consequently a destructor cannot sensibly be provided with a function `try` block as exceptions caught by a function `try` block's catch clause has already left the destructor's body (and will be retrown as with exceptions reaching a constructor's function `try` block's catch clause).

The consequences of an exception leaving the destructor's body is not defined, and may result in unexpected behavior. Consider the following example:

Assume a carpenter builds a cupboard containing a single drawer. The cupboard is finished, and a customer, buying the cupboard, finds that the cupboard can be used as expected. Satisfied with the cupboard, the customer asks the carpenter to build another cupboard, this time containing *two* drawers. When the second cupboard is finished, the customer takes it home and is utterly amazed when the second cupboard completely collapses immediately after its first use.

Weird story? Then consider the following program:

```
int main()
{
    try
    {
        cerr << "Creating Cupboard1\n";
        Cupboard1();
        cerr << "Beyond Cupboard1 object\n";
    }
    catch (...)
    {
        cerr << "Cupboard1 behaves as expected\n";
    }
    try
    {
        cerr << "Creating Cupboard2\n";
        Cupboard2();
        cerr << "Beyond Cupboard2 object\n";
    }
    catch (...)
    {
        cerr << "Cupboard2 behaves as expected\n";
    }
}
```

When this program is run it produces the following output:

```
Creating Cupboard1
Drawer 1 used
Cupboard1 behaves as expected
Creating Cupboard2
Drawer 2 used
Drawer 1 used
Abort
```

The final `Abort` indicates that the program has aborted instead of displaying a message like `Cupboard2 behaves as expected`.

Let's have a look at the three classes involved. The class `Drawer` has no particular characteristics, except that its destructor throws an exception:

```
class Drawer
{
    size_t d_nr;
    public:
        Drawer(size_t nr)
        :
            d_nr(nr)
        {}
        ~Drawer()
        {
```

```
            cerr << "Drawer " << d_nr << " used\n";
            throw 0;
        }
};
```

The class `Cupboard1` has no special characteristics at all. It merely has a single composed `Drawer` object:

```
class Cupboard1
{
    Drawer left;
    public:
        Cupboard1()
        :
            left(1)
        {}
};
```

The class `Cupboard2` is constructed comparably, but it has two composed `Drawer` objects:

```
class Cupboard2
{
    Drawer left;
    Drawer right;
    public:
        Cupboard2()
        :
            left(1),
            right(2)
        {}
};
```

When `Cupboard1`'s destructor is called, `Drawer`'s destructor is eventually called to destroy its composed object. This destructor throws an exception, which is caught beyond the program's first `try` block. This behavior is completely as expected.

Now a problem occurs when `Cupboard2`'s destructor is called. Of its two composed objects, the second `Drawer`'s destructor is called first. This destructor throws an exception, which ought to be caught beyond the program's second `try` block. However, although the flow of control by then has left the context of `Cupboard2`'s destructor, that object hasn't completely been destroyed yet as the destructor of its other (left) `Drawer` still has to be called.

Normally that would not be a big problem: once an exception is thrown from `Cupboard2`'s destructor any remaining actions would simply be ignored, albeit that (as both drawers are properly constructed objects) `left`'s destructor would still have to be called.

This happens here too and `left`'s destructor *also* needs to throw an exception. But as we've already left the context of the second `try` block, the current flow control is now thoroughly mixed up, and the program has no other option but to abort. It does so by calling `terminate()`, which in turn calls `abort()`. Here we have our collapsing cupboard having two drawers, even though the cupboard having one drawer behaves perfectly.

The program aborts since there are multiple composed objects whose destructors throw exceptions leaving the destructors. In this situation one of the composed objects would throw an exception by the time the program's flow control has already left its proper context causing the program to abort.

The **C++** standard therefore understandably stipulates that exceptions may *never* leave destructors. Here is the skeleton of a destructor whose code might throw exceptions. No function `try` block but all the destructor's actions are encapsulated in a `try` block nested under the destructor's body.

```
Class::~Class()
{
    try
    {
        maybe_throw_exceptions();
    }
    catch (...)
    {}
}
```

Chapter 10

More Operator Overloading

Having covered the overloaded assignment operator in chapter 8, and having shown several examples of other overloaded operators as well (i.e., the insertion and extraction operators in chapters 3 and 6), we will now take a look at operator overloading in general.

10.1 Overloading 'operator[]()'

As our next example of operator overloading, we introduce a class `IntArray` encapsulating an array of `int`s. Indexing the array elements is possible using the standard array index operator `[]`, but additionally checks for array bounds overflow will be performed. Furthermore, the index operator (`operator[]`) is interesting in that it can be used in expressions as both lvalue and as rvalue.

Here is an example showing the basic use of the class:

```
int main()
{
    IntArray x(20);                 // 20 ints

    for (int i = 0; i < 20; i++)
        x[i] = i * 2;               // assign the elements

    for (int i = 0; i <= 20; i++)   // produces boundary overflow
        cout << "At index " << i << ": value is " << x[i] << '\n';
}
```

First, the constructor is used to create an object containing 20 `int`s. The elements stored in the object can be assigned or retrieved. The first `for`-loop assigns values to the elements using the index operator, the second `for`-loop retrieves the values but will also result in a run-time error once the non-existing value `x[20]` is addressed. The `IntArray` class interface is:

```
class IntArray
{
    int     *d_data;
    size_t d_size;

     public:
        IntArray(size_t size = 1);
        IntArray(IntArray const &other);
        ~IntArray();
        IntArray const &operator=(IntArray const &other);
```

```
                                            // overloaded index operators:
        int &operator[](size_t index);                  // first
        int const &operator[](size_t index) const;      // second

        void swap(IntArray &other);         // trivial

    private:
        void boundary(size_t index) const;
        int &operatorIndex(size_t index) const;
};
```

This class has the following characteristics:

- One of its constructors has a `size_t` parameter having a default argument value, specifying the number of `int` elements in the object.
- The class internally uses a pointer to reach allocated memory. Hence, the necessary tools are provided: a copy constructor, an overloaded assignment operator and a destructor.
- Note that there are two overloaded index operators. Why are there two of them ?

  The first overloaded index operator allows us to reach and modify the elements of non-constant `IntArray` objects. This overloaded operator has as its prototype a function that returns *a reference* to an `int`. This allows us to use expressions like `x[10]` as rvalues *or* lvalues.

  With non-const `IntArray` objects `operator[]` can therefore be used to retrieve and to assign values. Note that the return value of the non-const `operator[]` member is *not* an `int const &`, but an `int &`. In this situation we don't use `const`, as we must be able to modify the element we want to access when the operator is used as lvalue.

  However, this whole scheme fails if there's nothing to assign. Consider the situation where we have an `IntArray const stable(5)`. Such an object is a *const* object which cannot be modified. The compiler detects this and will refuse to compile this object definition if only the non-const `operator[]` is available. Hence the second overloaded index operator. Here the return value is an `int const &`, rather than an `int &`, and the member function itself is a `const` member function. This second form of the overloaded index operator is only used with `const` objects. It is used for value *retrieval* instead of value assignment. That, of course, is precisely what we want when using `const` objects. Here, members are overloaded only by their `const` attribute. This form of function overloading was introduced earlier in the Annotations (sections 2.5.4 and 7.5).

  `IntArray`'s `operator[] const`, since `IntArray` stores values of a primitive type, could also define a value return type. However, with objects one usually doesn't want the extra copying that's implied with value return types. In those cases `const &` return values are preferred for `const` member functions. So, in the `IntArray` class an `int` return value could have been used as well, resulting in the following prototype:

  ```
  int IntArray::operator[](int index) const;
  ```

- As there is only one pointer data member, the destruction of the memory allocated by the object is a simple `delete[] data`.

Now, the implementation of the members (omitting the trivial implementation of `swap`, cf. chapter 8) are:

```
#include "intarray.ih"

IntArray::IntArray(size_t size)
:
    d_size(size)
{
```

```
        if (d_size < 1)
            throw string("IntArray: size of array must be >= 1");

        d_data = new int[d_size];
    }

    IntArray::IntArray(IntArray const &other)
    :
        d_size(other.d_size);
        d_data(new int[d_size]);
    {
        memcpy(d_data, other.d_data, d_size * sizeof(int));
    }

    IntArray::~IntArray()
    {
        delete[] d_data;
    }

    IntArray const &IntArray::operator=(IntArray const &other)
    {
        IntArray tmp(other);
        swap(other);
        return *this;
    }

    int &IntArray::operatorIndex(size_t index) const
    {
        boundary(index);
        return d_data[index];
    }

    int &IntArray::operator[](size_t index)
    {
        return operatorIndex(index);
    }

    int const &IntArray::operator[](size_t index) const
    {
        return operatorIndex(index);
    }

    void IntArray::boundary(size_t index) const
    {
        if (index < d_size)
            return;
        ostringstream out;
        out  << "IntArray: boundary overflow, index = " <<
                index << ", should be < " << d_size << '\n';
        throw out.str();
    }
```

Note how the `operator[]` members were implemented: as non-const members may call const member functions and as the implementation of the `const` member function is identical to the non-const member function's implementation both `operator[]` members could be defined inline using an auxiliary function `int &operatorIndex(size_t index) const`. A `const` member function may return a non-const reference (or pointer) return value, referring to one of the data members of its object. This is a potentially dangerous backdoor breaking data hiding. However, as the members in the public interface prevents this breach and so both public `operator[]` members may safely call the same `int`

`&operatorIndex() const` member, defining a *private backdoor*.

10.2 Overloading the insertion and extraction operators

Classes may be adapted in such a way that their objects may be inserted into and extracted from, respectively, a `std::ostream` and `std::istream`.

The class `std::ostream` defines insertion operators for primitive types, such as `int`, `char *`, etc.. In this section we will learn how to extend the existing functionality of classes (in particular `std::istream` and `std::ostream`) in such a way that they can be used also in combination with classes developed much later in history.

In this section we will show how the insertion operator can be overloaded allowing the insertion of any type of object, say `Person` (see chapter 8), into an `ostream`. Having defined such an overloaded operator the following will be possible:

```
Person kr("Kernighan and Ritchie", "unknown", "unknown");

cout << "Name, address and phone number of Person kr:\n" << kr << '\n';
```

The statement `cout << kr` uses `operator<<`. This member function has two operands: an `ostream &` and a `Person &`. The required action is defined in an overloaded *free function* `operator<<` expecting two arguments:

```
                            // declared in 'person.h'
std::ostream &operator<<(std::ostream &out, Person const &person);

                            // defined in some source file
ostream &operator<<(ostream &out, Person const &person)
{
    return
        out <<
            "Name:    " << person.name() << ", "
            "Address: " << person.address() << ", "
            "Phone:   " << person.phone();
}
```

The free function `operator<<` has the following noteworthy characteristics:

- The function returns a reference to an `ostream` object, to enable 'chaining' of the insertion operator.
- The two operands of `operator<<` are passed to the free function as its arguments. In the example, the parameter `out` was initialized by `cout`, the parameter `person` by `kr`.

In order to overload the *extraction* operator for, e.g., the `Person` class, members are needed modifying the class's private data members. Such *modifiers* are normally offered by the class interface. For the `Person` class these members could be the following:

```
void setName(char const *name);
void setAddress(char const *address);
void setPhone(char const *phone);
```

These members may easily be implemented: the memory pointed to by the corresponding data member must be deleted, and the data member should point to a copy of the text pointed to by the parameter. E.g.,

```
void Person::setAddress(char const *address)
{
    delete[] d_address;
    d_address = strdupnew(address);
}
```

A more elaborate function should check the reasonableness of the new address (`address` also shouldn't be a 0-pointer). This however, is not further pursued here. Instead, let's have a look at the final `operator>>`. A simple implementation is:

```
istream &operator>>(istream &in, Person &person)
{
    string name;
    string address;
    string phone;

    if (in >> name >> address >> phone)    // extract three strings
    {
        person.setName(name.c_str());
        person.setAddress(address.c_str());
        person.setPhone(phone.c_str());
    }
    return in;
}
```

Note the stepwise approach that is followed here. First, the required information is extracted using available extraction operators. Then, if that succeeds, *modifiers* are used to modify the data members of the object to be extracted. Finally, the stream object itself is returned as a reference.

10.3 Conversion operators

A class may be constructed around a built-in type. E.g., a class `String`, constructed around the `char *` type. Such a class may define all kinds of operations, like assignments. Take a look at the following class interface, designed after the `string` class:

```
class String
{
    char *d_string;

    public:
        String();
        String(char const *arg);
        ~String();
        String(String const &other);
        String const &operator=(String const &rvalue);
        String const &operator=(char const *rvalue);
};
```

Objects of this class can be initialized from a `char const *`, and also from a `String` itself. There is an overloaded assignment operator, allowing the assignment from a `String` object and from a `char const *`[1].

Usually, in classes that are less directly linked to their data than this `String` class, there will be an *accessor member function*, like a member `char const *String::c_str() const`. However, the

[1]Note that the assignment from a `char const *` also allows the null-pointer. An assignment like `stringObject = 0` is perfectly in order.

need to use this latter member doesn't appeal to our intuition when an array of `String` objects is defined by, e.g., a class `StringArray`. If this latter class provides the `operator[]` to access individual `String` members, it would most likely offer at least the following class interface:

```
class StringArray
{
    String *d_store;
    size_t d_n;

    public:
        StringArray(size_t size);
        StringArray(StringArray const &other);
        StringArray const &operator=(StringArray const &rvalue);
        ~StringArray();

        String &operator[](size_t index);
};
```

This interface allows us to assign `String` elements to each other:

```
StringArray sa(10);

sa[4] = sa[3];  // String to String assignment
```

But it is also possible to assign a `char const *` to an element of `sa`:

```
sa[3] = "hello world";
```

Here, the following steps are taken:

- First, `sa[3]` is evaluated. This results in a `String` reference.
- Next, the `String` class is inspected for an overloaded assignment, expecting a `char const *` to its right-hand side. This operator is found, and the string object `sa[3]` receives its new value.

Now we try to do it the other way around: how to *access* the `char const *` that's stored in `sa[3]`? The following attempt fails:

```
char const *cp = sa[3];
```

It fails since we would need an overloaded assignment operator for the 'class `char const *`'. Unfortunately, there isn't such a class, and therefore we can't build that overloaded assignment operator (see also section 10.12). Furthermore, *casting* won't work as the compiler doesn't know how to cast a `String` to a `char const *`. How to proceed?

One possibility is to define an accessor member function `c_str()`:

```
char const *cp = sa[3].c_str()
```

This compiles fine but looks clumsy.... A far better approach would be to use a *conversion operator*.

A *conversion operator* is a kind of overloaded operator, but this time the overloading is used to cast the object to another type. In class interfaces, the general form of a conversion operator is:

```
operator <type>() const;
```

Conversion operators usually are `const` member functions: they are automatically called when their objects are used as *rvalues* in expressions having an `type` *lvalue*. Using a conversion operator a `String` object may be interpreted as a `char const *` rvalue, allowing us to perform the above assignment.

Conversion operators are somewhat dangerous. The conversion is automatically performed by the compiler and unless its use is perfectly transparent it may confuse those who read code in which conversion operators are used. E.g., novice **C++** programmers are frequently confused by statements like '`if (cin) ...`'.

As a rule of thumb: classes should define at most one conversion operator. Multiple conversion operators may be defined but frequently result in ambiguous code. E.g., if a class defines `operator bool() const` and `operator int() const` then passing an object of this class to a function expecting a `size_t` argument results in an ambiguity as an `int` and a `bool` may both be used to initialize a `size_t`.

In the current example, the class `String` could define the following conversion operator for `char const *`:

```
String::operator char const *() const
{
    return d_string;
}
```

Notes:

- Conversion operators do not define return types. The conversion operator returns a value of the type specified beyond the `operator` keyword.
- In certain situations (e.g., when a `String` argument is passed to a function specifying an ellipsis parameter) the compiler needs a hand to disambiguate our intentions. A `static_cast` will solve the problem.
- With *template functions* conversion operators may not work immediately as expected. For example, when defining a conversion operator `X::operator std::string const() const` then `cout << X()` won't compile. The reason for this is explained in section 20.8, but a shortcut allowing the conversion operator to work is to define the following overloaded `operator<<` function:

```
std::ostream &operator<<(std::ostream &out, std::string const &str)
{
    return out.write(str.data(), str.length());
}
```

Conversion operators are also used when objects of classes defining conversion operators are inserted into streams. Realize that the right hand sides of insertion operators are function parameters that are initialized by the operator's right hand side arguments. The rules are simple:

- If a class `X` defining a conversion operator also defines an insertion operator accepting an `X` object the insertion operator is used;
- Otherwise, if the type returned by the conversion operator is insertable then the conversion operator is used;
- Otherwise, a compilation error results. Note that this happens if the type returned by the conversion operator itself defines a conversion operator to a type that may be inserted into a stream.

In the following example an object of class `Insertable` is directly inserted; an object of the class `Convertor` uses the conversion operator; an object of the class `Error` cannot be inserted since it does not define an insertion operator and the type returned by its conversion operator cannot be inserted

either (`Text` *does* define an `operator int() const`, but the fact that a `Text` itself cannot be inserted causes the error):

```
#include <iostream>
#include <string>
using namespace std;

struct Insertable
{
    operator int() const
    {
        cout << "op int()\n";
    }
};
ostream &operator<<(ostream &out, Insertable const &ins)
{
    return out << "insertion operator";
}
struct Convertor
{
    operator Insertable() const
    {
        return Insertable();
    }
};
struct Text
{
    operator int() const
    {
        return 1;
    }
};
struct Error
{
    operator Text() const
    {
        return Text();
    }
};

int main()
{
    Insertable insertable;
    cout << insertable << '\n';
    Convertor convertor;
    cout << convertor << '\n';
    Error error;
    cout << error << '\n';
}
```

Some final remarks regarding conversion operators:

- A conversion operator should be a 'natural extension' of the facilities of the object. For example, the stream classes define `operator bool()`, allowing constructions like `if (cin)`.
- A conversion operator should return an *rvalue*. It should do so to enforce data-hiding and because it is the intended use of the conversion operator. Defining a conversion operator as an *lvalue* (e.g., defining an `operator int &()` conversion operator) opens up a back door, and the operator can only be used as *lvalue* when explicitly called (as in: `x.operator int&() = 5`). Don't use it.

- Conversion operators should be defined as `const` member functions as they don't modify their object's data members.

- Conversion operators returning composed objects should return const references to these objects whenever possible to avoid calling the composed object's copy constructor.

10.4 The keyword 'explicit'

Conversions are not only performed by conversion operators, but also by constructors accepting one argument (i.e., constructors having one or multiple parameters, specifying default argument values for all parameters or for all but the first parameter).

Assume a data base class `DataBase` is defined in which `Person` objects can be stored. It defines a `Person *d_data` pointer, and so it offers a copy constructor and an overloaded assignment operator.

In addition to the copy constructor `DataBase` offers a default constructor and several additional constructors:

- `DataBase(Person const &)`: the `DataBase` initially contains a single `Person` object;

- `DataBase(istream &in)`: the data about multiple persons are read from `in`.

- `DataBase(size_t count, istream &in = cin)`: the data of `count` persons are read from `in`, by default the standard input stream.

The above constructors all are perfectly reasonable. But they also allow the compiler to compile the following code without producing any warning at all:

```
DataBase db;
DataBase db2;
Person person;

db2 = db;           // 1
db2 = person;       // 2
db2 = 10;           // 3
db2 = cin;          // 4
```

Statement 1 is perfectly reasonable: `db` is used to redefine `db2`. Statement 2 might be understandable since we designed `DataBase` to contain `Person` objects. Nevertheless, we might question the logic that's used here as a `Person` is not some kind of `DataBase`. The logic becomes even more opaque when looking at statements 3 and 4. Statement 3 in effect will wait for the data of 10 persons to appear at the standard input stream. Nothing like that is suggested by `db2 = 10`.

All four statements are the result of implicit promotions. Since constructors accepting, respectively a `Person`, a `size_t` and an `istream` have been defined for `PersonData` and since the assignment operator expects a `PersonData` right-hand side (rhs) argument the compiler will first convert the rhs arguments to anonymous `DataBase` objects which are then assigned to `db2`.

It is good practice to prevent implicit promotions by using the `explicit` modifier when declaring a constructor. Constructors using the `explicit` modifier can only be used to construct objects explicitly. Statements 1-4 would not have compiled if the constructors expecting one argument would have been declared using `explicit`. E.g.,

```
explicit DataBase(Person const &person);
explicit DataBase(size_t count, std:istream &in);
```

Having declared all constructors accepting one argument as `explicit` the above assignments would have required the explicit specification of the appropriate constructors, thus clarifying the programmer's intent:

```
DataBase db;
DataBase db2;
Person person;

db2 = db;               // 1
db2 = DataBase(person); // 2
db2 = DataBase(10);     // 3
db2 = DataBase(cin);    // 4
```

As a rule of thumb prefix one argument constructors with the `explicit` keyword unless implicit promotions are perfectly natural (`string`'s `char const *` accepting constructor is a case in point).

10.4.1 Explicit conversion operators (C++0x, ?)

In addition to explicit constructors, the C++0x standard adds *explicit conversion operators* to **C++**.

For example, a class might define `operator bool() const` returning `true` if an object of that class is in a usable state and `false` if not. Since the type `bool` is an arithmetic type this could result in unexpected or unintended behavior. Consider:

```
class StreamHandler
{
    public:
        operator bool() const;      // true: object is fit for use
        ...
};

int fun(StreamHandler &sh)
{
    int sx;

    if (sh)                         // intended use of operator bool()
        ... use sh as usual; also use 'sx'

    process(sh);                    // typo: 'sx' was intended
}
```

In this example `process` unintentionally receives the value returned by `operator bool` using the implicit conversion from `bool` to `int`.

With `explicit` conversion operators implicit conversions like the one shown in the example are prevented and such conversion operators will only be used in situations where the converted type is explicitly required. E.g, in the condition sections of `if` or repetition statements a `bool` value is expected. In such cases an `explicit operator bool()` converion operator would automatically be used.

10.5 Overloading the increment and decrement operators

Overloading the increment operator (`operator++`) and decrement operator (`operator--`) introduces a small problem: there are two version of each operator, as they may be used as *postfix operator* (e.g., `x++`) or as *prefix operator* (e.g., `++x`).

Used as *postfix* operator, the value's object is returned as an *rvalue*, temporary const object and the post-incremented variable itself disappears from view. Used as *prefix* operator, the variable is incremented, and its value is returned as *lvalue* and it may be altered again by modifying the prefix operator's return value. Whereas these characteristics are not *required* when the operator is overloaded, it is strongly advised to implement these characteristics in any overloaded increment or decrement operator.

Suppose we define a *wrapper class* around the `size_t` value type. Such a class could offer the following (partially shown) interface:

```
class Unsigned
{
    size_t d_value;

    public:
        Unsigned();
        explicit Unsigned(size_t init);

        Unsigned &operator++();
}
```

The class's last member declares the *prefix* overloaded increment operator. The returned *lvalue* is `Unsigned &`. The member is easily implemented:

```
Unsigned &Unsigned::operator++()
{
    ++d_value;
    return *this;
}
```

To define the *postfix* operator, an overloaded version of the operator is defined, expecting a (dummy) `int` argument. This might be considered a *kludge*, or an acceptable application of function overloading. Whatever your opinion in this matter, the following can be concluded:

- Overloaded increment and decrement operators *without parameters* are *prefix* operators, and should return *references* to the current object.
- Overloaded increment and decrement operators *having an int parameter* are *postfix* operators, and should return a constant value which is a copy of the object at the point where its postfix operator is used.

The postfix increment operator is declared as follows in the class `Unisigned`'s interface:

```
Unsigned const operator++(int);
```

It may be implemented as follows:

```
Unsigned const Unsigned::operator++(int)
{
    Unsigned tmp(*this);
    ++d_value;
    return tmp;
}
```

Note that the operator's parameter is not used. It is only part of the implementation to *disambiguate* the prefix- and postfix operators in implementations and declarations.

In the above example the statement incrementing the current object offers the *nothrow* guarantee as it only involves an operation on a primitive type. If the initial copy construction throws then the

original object is not modified, if the return statement throws the object has safely been modified. But incrementing an object could itself throw exceptions. How to implement the increment operators in that case? Once again, `swap` is our friend. Here are the pre- and postfix operators offering the strong guarantee when the member `increment` performing the increment operation may throw:

```
Unsigned &Unsigned::operator++()
{
    Unsigned tmp(*this);
    tmp.increment();
    swap(tmp);
    return *this;
}
Unsigned const Unsigned::operator++(int)
{
    Unsigned tmp(*this);
    tmp.increment();
    swap(tmp);
    return tmp;
}
```

The postfix increment operator first creates a copy of the current object. That copy is incrmented and then swapped with the current object. If `increment` throws the current object remains unaltered; the swap operation ensures that the original object is returned and the current object becomes the incremented object.

When calling the postfix increment or decrement operator using its full member function name then any `int` argument passed to the member function will result in calling the postfix operator. Example:

```
Unsigned uns(13);

uns.operator++();     // prefix-incrementing uns
uns.operator++(0);    // postfix-incrementing uns
```

10.6 Overloading binary operators

In various classes overloading binary operators (like `operator+`) can be a very natural extension of the class's functionality. For example, the `std::string` class has various overloaded forms of `operator+`.

Most binary operators come in two flavors: the plain binary operator (like the + operator) and the arithmetic assignment variant (like the += operator). Whereas the plain binary operators return const expression values, the arithmetic assignment operators return a (non-const) reference to the object to which the operator was applied. For example, with `std::string` objects the following code (annotations below the example) may be used:

```
std::string s1;
std::string s2;
std::string s3;

s1 = s2 += s3;                  // 1
(s2 += s3) + " postfix";        // 2
s1 = "prefix " + s3;            // 3
"prefix " + s3 + "postfix";     // 4
("prefix " + s3) += "postfix";  // 5
```

- at `// 1` the contents of `s3` is added to `s2`. Next, `s2` is returned, and its new contents are assigned to `s1`. Note that += returns `s2` itself.

- at `// 2` the contents of `s3` is also added to `s2`, but as `+=` returns `s2` itself, it's possible to add some more to `s2`
- at `// 3` the `+` operator returns a `std::string` containing the concatenation of the text `prefix` and the contents of `s3`. This string returned by the `+` operator is thereupon assigned to `s1`.
- at `// 4` the `+` operator is applied twice. The effect is:
  1. The first `+` returns a `std::string` containing the concatenation of the text `prefix` and the contents of `s3`.
  2. The second `+` operator takes this returned string as its left hand value, and returns a string containing the concatenated text of its left and right hand operands.
  3. The string returned by the second `+` operator represents the value of the expression.
- statement `// 5` is interesting in that it should not compile but does compile with the `Gnu` compiler. It should not compile, as the `+` operator should return a `const` string, thereby preventing its modification by the subsequent `+=` operator (compare this situation with `int` values: `(3 + intVar) += 4` won't compile. Clearly, `std::string` does *not* do 'as the ints do'). However, in the Annotations we will consequently let binary arithmetic operators return `const` values.

Consider the following code, in which a class `Binary` supports an overloaded `operator+`:

```
class Binary
{
    public:
        Binary();
        Binary(int value);
        Binary const operator+(Binary const &rvalue);
};

int main()
{
    Binary b1;
    Binary b2(5);

    b1 = b2 + 3;            // 1
    b1 = 3 + b2;            // 2
}
```

Compilation of this little program fails for statement `// 2`, with the compiler reporting an error like:

```
error: no match for 'operator+' in '3 + b2'
```

Why is statement `// 1` compiled correctly whereas statement `// 2` won't compile?

In order to understand this remember *promotions*. As we have seen in section 10.4, constructors expecting a single argument may be implicitly activated when an argument of an appropriate type is provided. We've encountered this repeatedly with `std::string` objects, where an `ASCII-Z` string may be used to initialize a `std::string` object.

Analogously, in statement `// 1`, the `+` operator is called for the `b2` object. This operator expects another `Binary` object as its right hand operand. However, an `int` is provided. As a constructor `Binary(int)` exists, the `int` value is first promoted to a `Binary` object. Next, this `Binary` object is passed as argument to the `operator+` member.

In statement `// 2` no promotions are available: here the `+` operator is applied to an lvalue that is an `int`. An `int` is a primitive type and primitive types have no concept of 'constructors', 'member functions' or 'promotions'.

How, then, are promotions of left-hand operands implemented in statements like `"prefix " + s3`? Since promotions are applied to function arguments, we must make sure that both operands of binary operators are arguments. This means that plain binary arithmetic operators should be declared as *free operators*, also called *free functions*. Functions like the plain binary arithmetic operators conceptually belong to the class for which they implement the binary operator. Consequently they should be declared *within* the class's header file. We will cover their implementations shortly, but here is our first revision of the declaration of the class `Binary`, declaring an overloaded + operator as a free function:

```
class Binary
{
    public:
        Binary();
        Binary(int value);
};

Binary const operator+(Binary const &l_hand, Binary const &r_hand);
```

By defining binary operators as free functions, the following promotions are possible:

- If the left-hand operand is of the intended class type, the right hand argument will be promoted whenever possible;
- If the right-hand operand is of the intended class type, the left hand argument will be promoted whenever possible;
- No promotions occur when none of the operands are of the intended class type;
- An ambiguity occurs when promotions to different classes are possible for the two operands. For example:

```
class A;

class B
{
    public:
        B(A const &a);
};

class A
{
    public:
        A();
        A(B const &b);
};

A const operator+(A const &a, B const &b);
B const operator+(B const &b, A const &a);

int main()
{
    A a;

    a + a;
};
```

Here, both overloaded + operators are possible when compiling the statement `a + a`. The ambiguity must be solved by explicitly promoting one of the arguments, e.g., `a + B(a)` will allow the compiler to resolve the ambiguity to the first overloaded + operator.

The next step is to implement the corresponding overloaded arithmetic assignment operator. As this operator *always* has a left-hand operand which is an object of its own class, it is implemented as a true member function. Furthermore, the arithmetic assignment operator should return a reference to the object to which the arithmetic operation applies, as the object might be modified in the same statement. E.g., `(s2 += s3) + " postfix"`. Here is our second revision of the class `Binary`, showing both the declaration of the plain binary operator and the corresponding arithmetic assignment operator:

```
class Binary
{
    public:
        Binary();
        Binary(int value);
        Binary const operator+(Binary const &rvalue);

        Binary &operator+=(Binary const &other);
};

Binary const operator+(Binary const &lhs, Binary const &rhs);
```

If a class benefits from being move-aware then for each `const` reference parameter an overloaded function expecting a `const` rvalue reference may be considered. With the class `Binary` and its `operator+` this would boild down to the following overloaded `operator+` functions:

```
Binary const operator+(Binary const &&ltmp, Binary const &rhs);
Binary const operator+(Binary const &lhs,   Binary const &&rtmp);
Binary const operator+(Binary const &&ltmp, Binary const &rtmp);
```

When implementing the arithmetic assignment operator the strong guarantee should again be kept in mind. Use a temporary object and swap if the arithmetic operation might throw. Example:

```
Binary &operator+=(Binary const &other)
{
    Binary tmp(*this);
    tmp.add(other);     // this may throw
    swap(tmp);
    return *this;
}
```

Here, too, defining a `operator+=(Binary const &&tmp)` should be considered if `Binary` shold be a move-aware class.

Having available the arithmetic assignment operator, implementating the plain binary operator is easy. The `lhs` argument is copied into a `Binary tmp` to which the `rhs` operand is added. Then `tmp` is returned. The copy construction and two statements may be contracted into one single return statement, but then compilers usually aren't able to apply copy elision resulting in two copy constructions. Usually copy elision is performed when the steps are taken separately:

```
class Binary
{
    public:
        Binary();
        Binary(int value);
        Binary const operator+(Binary const &rvalue);

        Binary &operator+=(Binary const &other);
};
```

```
Binary const operator+(Binary const &lhs, Binary const &rhs)
{
    Binary tmp(lhs);
    tmp += ths;
    return tmp;
}
```

10.7 Overloading 'operator new(size_t)'

When `operator new` is overloaded, it must define a `void *` return type, and its first parameter must be of type `size_t`. The default `operator new` defines only one parameter, but overloaded versions may define multiple parameters the first of not explicitly specified but is deducted from the size of the object of the class for which `operator new` is overloaded. In this section overloading `operator new` is discussed. Overloading `new[]` is discussed in section 10.9.

It is possible to define multiple versions of the `operator new`, as long as each version defines its own unique set of arguments. When overloaded `operator new` members must dynamically allocate memory they can do so using the global `operator new` using the scope resolution operator `::`. In the next example the overloaded `operator new` of the class `String` will initialize the substrate of dynamically allocated `String` objects to 0-bytes:

```
#include <cstddef>
#include <iosfwd>

class String
{
    std::string *d_data;

    public:
        void *operator new(size_t size)
        {
            return memset(::operator new(size), 0, size);
        }
        bool empty() const
        {
            return d_data == 0;
        }
};
```

The above `operator new` is used in the following program, illustrating that even though `String`'s default constructor does nothing the object's data are initialized to zeroes:

```
#include "string.h"
#include <iostream>
using namespace std;

int main()
{
    String *sp = new String;

    cout << boolalpha << sp->empty() << '\n';   // shows: true
}
```

At `new String` the following took place:

- First, `String::operator new` was called, allocating and initializing a block of memory, the size of a `String` object.

- Next, a pointer to this block of memory was passed to the (default) `String` constructor. Since no constructor was defined, the constructor itself didn't do anything at all.

As `String::operator new` initialized the allocated memory to zero bytes the allocated `String` object's `d_data` member had already been initialized to a 0-pointer by the time it started to exist.

All member functions (including constructors and destructors) we've encountered so far define a (hidden) pointer to the object on which they should operate. This hidden pointer becomes the function's `this` pointer.

In the next example of *pseudo* **C++** *code*, the pointer is explicitly shown to illustrate what's happening when `operator new` is used. In the first part an `String` object `str` is defined directly, in the second part of the example the (overloaded) `operator new` is used:

```
String::String(String *const this);     // real prototype of the default
                                        // constructor

String *sp = new String;                // This statement is implemented
                                        // as follows:

    String *sp = reinterpret_cast<String *>(            // allocation
                    String::operator new(sizeof(String))
                 );
    String::String(sp);                                 // initialization
```

In the above fragment the member functions were treated as an *object-less* member function of the class `String`. Such members are called *static member functions* that are covered in chapter 11. Actually, `operator new` *is* such a static member function. Since it has no `this` pointer it cannot reach data members of the object for which it is expected to make memory available. It can only allocate and initialize the allocated memory, but cannot reach the object's data members by name as there is as yet no data object layout defined.

Following the allocation, the memory is passed (as the `this` pointer) to the constructor for further processing.

`Operator new` can have multiple parameters. The first parameter is initialized as an implicit argument and is always a `size_t` parameter. Additional overloaded operators may defined additional parameters. An interesting additional `operator new` is the *placement new* operator. With the placement new operator a block of memory has already been set aside and one of the class's constructors will be used to initialize that memory. Overloading placement new requires an `operator new` having two parameters: `size_t` and `char *`, pointing to the memory that was already available. The `size_t` parameter is implicitly initialized, but the remaining parameters must explicitly be initialized using arguments to `operator new`. Hence we reach the familiar syntactical form of the placement new operator in use:

```
char buffer[sizeof(String)];        // predefined memory
String *sp = new(buffer) String;    // placement new call
```

The declaration of the placement new operator in our class `String` looks like this:

```
void *operator new(size_t size, char *memory);
```

It could be implemented like this (also initializing the `String`'s memory to 0-bytes):

```
void *String::operator new(size_t size, char *memory)
{
    return memset(memory, 0, size);
}
```

Any other overloaded version of `operator new` could also be defined. Here is an example showing the use and definition of an overloaded `operator new` storing the object's address immediately in an existing array of pointers to `String` objects (assuming the array is large enough):

```
    // use:
String *next(String **pointers, size_t *idx)
{
    return new(pointers, (*idx)++) String;
}

    // implementation:
void *String::operator new(size_t size, char **pointers, size_t idx)
{
    return pointers[idx] = ::operator new(size);
}
```

10.8 Overloading 'operator delete(void *)'

The `delete` operator may also be overloaded. In fact it's good practice to overload `operator delete` whenever `operator new` is also overloaded.

`Operator delete` must define a `void *` parameter. A second overloaded version defining a second parameter of type `size_t` is related to overloading `operator new[]` and is discussed in section 10.9.

Overloaded `operator delete` members return `void`.

The 'home-made' `operator delete` is called when deleting a dynamically allocated object after executing the destructor of the associated class. So, the statement

```
delete ptr;
```

with `ptr` being a pointer to an object of the class `String` for which the operator `delete` was overloaded, is a shorthand for the following statememnts:

```
ptr->~String(); // call the class's destructor

                // and do things with the memory pointed to by ptr
String::operator delete(ptr);
```

The overloaded `operator delete` may do whatever it wants to do with the memory pointed to by `ptr`. It could, e.g., simply delete it. If that would be the preferred thing to do, then the default `delete` operator can be called using the `::` scope resolution operator. For example:

```
void String::operator delete(void *ptr)
{
    // any operation considered necessary, then, maybe:
    ::delete ptr;
}
```

To declare the above overloaded `operator delete` simply add the following line to the class's interface:

```
void operator delete(void *ptr);
```

Like `operator new operator delete` is a static member function (see also chapter 11).

10.9 Operators 'new[]' and 'delete[]'

In sections 8.1.1, 8.1.2 and 8.2.1 `operator new[]` and `operator delete[]` were introduced. Like `operator new` and `operator delete` the operators `new[]` and `delete[]` may be overloaded.

As it is possible to overload `new[]` and `delete[]` as well as `operator new` and `operator delete`, one should be careful in selecting the appropriate set of operators. The following rule of thumb should always be applied:

> If `new` is used to allocate memory, `delete` should be used to deallocate memory. If `new[]` is used to allocate memory, `delete[]` should be used to deallocate memory.

By default these operators act as follows:

- `operator new` is used to allocate a single object or primitive value. With an object, the object's constructor is called.
- `operator delete` is used to return the memory allocated by `operator new`. Again, with class-type objects, the class's destructor is called.
- `operator new[]` is used to allocate a series of primitive values or objects. If a series of objects is allocated, the class's default constructor is called to initialize each object individually.
- `operator delete[]` is used to delete the memory previously allocated by `new[]`. *If* objects were previously allocated, then the destructor will be called for each individual object. Be careful, though, when poiners to objects were allocated. If *pointers to objects* were allocated the destructors of the objects to which the allocated pointers point won't automatically be called. A pointer is a primitive type and so no further action is taken when it is returned to the common pool.

10.9.1 Overloading 'new[]'

To overload `operator new[]` in a class (e.g., in the class `String`) add the following line to the class's interface:

```
void *operator new[](size_t size);
```

The member's `size` parameter is implicitly provided and is initialized bij **C++**'s run-time system to the amount of memory that must be allocated. Like the simple one-object `operator new` it should return a `void *`. The number of objects that must be initialized can easily be computed from `size / sizeof(String)` (and of course replacing `String` by the appropriate class name when overloading `operator new[]` for another class). The overloaded `new[]` member may allocate raw memory using e.g., the default `operator new[]` or the default `operator new`:

```
void *operator new[](size_t size)
{
    return ::operator new[](size);
    // alternatively:
    // return ::operator new(size);
}
```

Before returning the allocated memory the overloaded `operator new[]` has a chance to do something special. It could, e.g., initialize the memory to zero-bytes.

Once the overloaded `operator new[]` has been defined, it will be used automatically in statements like:

```
String *op = new String[12];
```

Like `operator new` additional overloads of `operator new[]` may be defined. One opportunity for an `operator new[]` overload is overloading *placement new* specifically for arrays of objects. This operator is available by default but becomes unavailable once at least one overloaded `operator new[]` is defined. Implementing placement `new` is not difficult. Here is an example, initializing the available memory to 0-bytes before returning:

```
void *String::operator new[](size_t size, char *memory)
{
    return memset(memory, 0, size);
}
```

To use this overloaded operator, the second parameter must again be provided, as in:

```
char buffer[12 * sizeof(String)];
String *sp = new(buffer) String[12];
```

10.9.2 Overloading 'delete[]'

To overload `operator delete[]` in a class `String` add the following line to the class's interface:

```
void operator delete[](void *memory);
```

Its parameter is initialized to the address of a block of memory previously allocated by `String::new[]`.

There are some subtleties to be aware of when implementing `operator delete[]`. Although the addresses returned by `new` and `new[]` point to the allocated object(s), there is an additional `size_t` value available immediately before the address returned by `new` and `new[]`. This `size_t` value is part of the allocated block and contains the actual size of the block. This of course does not hold true for the placement `new` operator.

When a class defines a destructor the `size_t` value preceding the address returned by `new[]` does not contain the size of the allocated block, but the *number* of objects specified when calling `new[]`. Normally that is of no interest, but when overloading `operator delete[]` it might become a useful piece of information. In those cases `operator[]` will *not* receive the address returned by `new[]` but rather the address of the initial `size_t` value. Whether this is at all useful is not clear. By the time `delete[]`'s code is executed all objects have already been destroyed, so `operator delete[]` is only to determine how many objects were destroyed but the objects themselves cannot be used anymore.

Here is an example showing this behavior of `operator delete[]` for a minimal `Demo` class:

```
struct Demo
{
    size_t idx;
    Demo()
    {
        cout << "default cons\n";
    }
    ~Demo()
    {
        cout << "destructor\n";
    }
    void *operator new[](size_t size)
    {
        return ::operator new(size);
    }
    void operator delete[](void *vp)
    {
```

```
        cout << "delete[] for: " << vp << '\n';
        ::operator delete[](vp);
    }
};

int main()
{
    Demo *xp;
    cout << ((int *)(xp = new Demo[3]))[-1] << '\n';
    cout << xp << '\n';
    cout << "==================\n";
    delete[] xp;
}
// This program displays (your 0x?????? addresses might differ, but
// the difference between the two should be sizeof(size_t)):
//  default cons
//  default cons
//  default cons
//  3
//  0x8bdd00c
//  ==================
//  destructor
//  destructor
//  destructor
//  delete[] for: 0x8bdd008
```

Having overloaded `operator delete[]` for a class `String`, it will be used automatically in statements like:

```
    delete[] new String[5];
```

Operator `delete[]` may also be overloaded using an additional `size_t` parameter:

```
void operator delete[](void *p, size_t size);
```

Here `size` is automatically initialized to the size (in bytes) of the block of memory to which `void *p` points. If this form is defined, then `void operator[](void *)` should *not* be defined, to avoid ambiguities. An example of this latter form of `operator delete[]` is:

```
void String::operator delete[](void *p, size_t size)
{
    cout << "deleting " << size << " bytes\n";
    ::operator delete[](ptr);
}
```

Additional overloads of `operator delete[]` may be defined, but to use them they must explicitly be called as static member functions (cf. chapter 11). Example:

```
    // declaration:
void String::operator delete[](void *p, ostream &out);
    // usage:
String *xp = new String[3];
String::operator delete[](xp, cout);
```

10.9.3 'new[]', 'delete[]' and exceptions

When an exception is thrown while executing a `new[]` expression, what will happen? In this section we'll show that `new[]` is exception safe even only some of the objects were properly constructed.

To begin, `new[]` might throw while trying to allocate the required memory. In this case a `bad_alloc` is thrown and we don't leak as nothing was allocated.

Having allocated the required memory the class's default constructor will be called for each of the objects in turn. At some point a constructor might throw. What happens next is defined by the **C++** standard: the destructors of the already constructed objects will be called and the memory allocated for the objects themselves is returned to the common pool. Assuming that the failing constructor offers the basic guarantee `new[]` is therefore exception safe even if a constructor may throw.

The following example illustrates this behavior. A request to allocate and initialize five objects is made, but after constructing two objects construction fails by throwing an exception. The output shows that the destructors of properly constructed objects are called and that the allocated *substrate memory* is properly returned:

```
#include <iostream>
using namespace std;

static size_t count = 0;

class X
{
    int x;

    public:
        X()
        {
            if (count == 2)
                throw 1;
            cout << "Object " << ++count << endl;
        }
        ~X()
        {
            cout << "Destroyed " << this << "\n";
        }
        void *operator new[](size_t size)
        {
            cout << "Allocating objects: " << size << " bytes\n";
            return ::operator new(size);
        }
        void operator delete[](void *mem)
        {
            cout << "Deleting memory at " << mem << ", containing: " <<
                *reinterpret_cast<int *>(mem) << "\n";
            ::operator delete(mem);
        }
};

int main()
try
{
    X *xp = new X[5];
    cout << "Memory at " << xp << endl;
    delete[] xp;
}
```

```
catch (...)
{
    cout << "Caught exception.\n";
}
// Output from this program (your 0x??? addresses might differ)
//  Allocating objects: 24 bytes
//  Object 1
//  Object 2
//  Destroyed 0x8428010
//  Destroyed 0x842800c
//  Deleting memory at 0x8428008, containing: 5
//  Caught exception.
```

10.10 Function Objects

Function Objects are created by overloading the *function call operator* `operator()`. By defining the function call operator an object masquerades as a function, hence the term *function objects*.

Function objects are important when using *generic algorithms*. The use of function objects is preferred over alternatives like pointers to functions. The fact that they are important in the context of generic algorithms leaves us with a didactic dilemma. At this point in the **C++** Annotations it would have been nice if generic algorithms would already have been covered, but for the discussion of the generic algorithms knowledge of function objects is required. This bootstrapping problem is solved in a well known way: by ignoring the dependency for the time being, for now concentrating on the function object concept.

Function objects are objects for which `operator()` has been defined. Function objects are not just used in combination with generic algorithms, but also as a (preferred) alternative to pointers to functions.

Function objects are frequently used to implement *predicate* functions. Predicate functions return boolean values. Predicate functions and predicate function objects are commonly referred to as 'predicates'. Predicates are frequently used by generic algorithms such as the count_if generic algorithm, covered in chapter 19, returning the number of times its function object has returned `true`. In the *standard template library* two kinds of predicates are used: *unary predicates* receive one argument, *binary predicates* receive two arguments.

Assume we have a class `Person` and an array of `Person` objects. Further assume that the array is not sorted. A well known procedure for finding a particular `Person` object in the array is to use the function **lsearch**(3), which performs a *lineair search* in an array. Example:

```
Person &target = targetPerson();    // determine the person to find
Person *pArray;
size_t n = fillPerson(&pArray);

cout << "The target person is";

if (!lsearch(&target, pArray, &n, sizeof(Person), compareFunction))
    cout << " not";
cout << "found\n";
```

The function `targetPerson` determines the person we're looking for, and `fillPerson` is called to fill the array. Then `lsearch` is used to locate the target person.

The comparison function must be available, as its address is one of the arguments of `lsearch`. It must be a real function having an address. If it is defined inline then the compiler has no choice but to ignore that request as inline functions don' have addresses. `CompareFunction` could be implemented like this:

```
int compareFunction(void const *p1, void const *p2)
{
    return *reinterpret_cast<Person const *>(p1)    // lsearch wants 0
            !=                                      // for equal objects
            *reinterpret_cast<Person const *>(p2);
}
```

This, of course, assumes that the `operator!=` has been overloaded in the class `Person`. But overloading `operator!=` is no big deal, so let's assume that that operator is actually available.

On average `n / 2` times *at least* the following actions take place:

1. The two arguments of the compare function are pushed on the stack;
2. The value of the final parameter of `lsearch` is determined, producing `compareFunction`'s address;
3. The compare function is called;
4. Then, inside the compare function the address of the right-hand argument of the `Person::operator!=` argument is pushed on the stack;
5. `Person::operator!=` is evaluated;
6. The argument of the `Person::operator!=` function is popped off the stack;
7. The two arguments of the compare function are popped off the stack.

Using function objects results in a different picture. Assume we have constructed a function `PersonSearch`, having the following prototype (this, however, is not the preferred approach. Normally a generic algorithm is preferred over a home-made function. But for now we focus on `PersonSearch` to illustrate the use and implementation of a function object):

```
Person const *PersonSearch(Person *base, size_t nmemb,
                           Person const &target);
```

This function can be used as follows:

```
Person &target = targetPerson();
Person *pArray;
size_t n = fillPerson(&pArray);

cout << "The target person is";

if (!PersonSearch(pArray, n, target))
    cout << " not";

cout << "found\n";
```

So far, not much has been changed. We've replaced the call to `lsearch` with a call to another function: `PersonSearch`. Now look at `PersonSearch` itself:

```
Person const *PersonSearch(Person *base, size_t nmemb,
                           Person const &target)
{
    for (int idx = 0; idx < nmemb; ++idx)
        if (target(base[idx]))
            return base + idx;
    return 0;
}
```

`PersonSearch` implements a plain linear search. However, in the for-loop we see `target(base[idx])`. Here `target` is used as a *function object*. Its implementation is simple:

```
bool Person::operator()(Person const &other) const
{
    return *this != other;
}
```

Note the somewhat peculiar syntax: `operator()`. The first set of parentheses define the operator that is overloaded: the function call operator. The second set of parentheses define the parameters that are required for this overloaded operator. In the class header file this overloaded operator is declared as:

```
bool operator()(Person const &other) const;
```

Clearly `Person::operator()` is a simple function. It contains but one statement, and we could consider defining it inline. Assuming we do, then this is what happens when `operator()` is called:

- The address of the right-hand argument of the `Person::operator!=` argument is pushed on the stack,
- The `operator!=` function is evaluated,
- The argument of `Person::operator!=` argument is popped off the stack,

Due to the fact that `operator()` is an inline function, it is not actually called. Instead `operator!=` is called immediately. Moreover, the required stack operations are fairly modest.

Function objects may truly be defined inline. Functions that are called indirectly (i.e., using pointers to functions) can never be defined inline as their addresses must be known. Therefore, even if the function object needs to do very little work it is defined as an ordinary function if it is going to be called through pointers. The overhead of performing the indirect call may annihilate the advantage of the flexibility of calling functions indirectly. In these cases using inline function objects can result in an increase of a program's efficiency.

An added benefit of function objects is that they may access the private data of their objects. In a search algorithm where a compare function is used (as with `lsearch`) the target and array elements are passed to the compare function using pointers, involving extra stack handling. Using function objects, the target person doesn't vary within a single search task. Therefore, the target person could be passed to the function object's class constructor. This is in fact what happens in the expression `target(base[idx])` receiving as its only argument the subsequent elements of the array to search.

10.10.1 Constructing manipulators

In chapter 6 we saw constructions like `cout << hex << 13 <<` to display the value 13 in hexadecimal format. One may wonder by what magic the `hex` manipulator accomplishes this. In this section the construction of manipulators like `hex` is covered.

Actually the construction of a manipulator is rather simple. To start, a definition of the manipulator is needed. Let's assume we want to create a manipulator `w10` which will set the field width of the next field to be written by the `ostream` object to 10. This manipulator is constructed as a function. The `w10` function will have to know about the `ostream` object in which the width must be set. By providing the function with an `ostream &` parameter, it obtains this knowledge. Now that the function knows about the `ostream` object we're referring to, it can set the width in that object.

Next, it must be possible to use the manipulator in an insertion sequence. This implies that the return value of the manipulator must be a reference to an `ostream` object also.

From the above considerations we're now able to construct our `w10` function:

```
#include <ostream>
#include <iomanip>

std::ostream &w10(std::ostream &str)
{
    return str << std::setw(10);
}
```

The `w10` function can of course be used in a 'stand alone' mode, but it can also be used as a manipulator. E.g.,

```
#include <iostream>
#include <iomanip>

using namespace std;

extern ostream &w10(ostream &str);

int main()
{
    w10(cout) << 3 << " ships sailed to America\n;
    cout << "And " << w10 << 3 << " more ships sailed too.\n";
}
```

The `w10` function can be used as a manipulator because the `class ostream` has an overloaded `operator<<` accepting a pointer to a function expecting an `ostream &` and returning an `ostream &`. Its definition is:

```
ostream& operator<<(ostream ostream &(*func)(ostream &str))
{
    return (*func)(*this);
}
```

In addition to the above overloaded `operator<<` another one is defined

```
ostream &operator<<(ios_base &(*func)(ios_base &base))
{
    (*func)(*this);
    return *this;
}
```

This latter function is used when inserting, e.g., `hex` or `internal`.

The above procedure does not work for manipulators requiring arguments. It is of course possible to overload `operator<<` to accept an `ostream` reference and the address of a function expecting an `ostream &` and, e.g., an `int`, but while the address of such a function may be specified with the <<-operator, the arguments itself cannot be specified. So, one wonders how the following construction has been implemented:

```
cout << setprecision(3)
```

In this case the manipulator is defined as a macro. Macro's, however, are the realm of the preprocessor, and may easily suffer from unwelcome side-effects. In **C++** programs they should be avoided whenever possible. The following section introduces a way to implement manipulators requiring arguments without resorting to macros, but using anonymous objects.

10.10.1.1 Manipulators requiring arguments

Manipulators taking arguments are implemented as macros: they are handled by the preprocessor, and are not available beyond the preprocessing stage. The problem appears to be that you can't call a function in an insertion sequence: when using multiple `operator<<` operators in one statement the compiler will call the functions, save their return values, and will then use their return values in the insertion sequence. That invalidates the ordering of the arguments passed to your <<-operators.

So, one might consider constructing another overloaded `operator<<` accepting the address of a function receiving not just the `ostream` reference, but a series of other arguments as well. But this creaes the problem that it isn't clear how the function should receive its arguments: you can't just call it since that takes us back to the abovementioned problem. Merely passing its address is fine, but then no arguments can't be passed to the function.

There exists a solution, based on the use of anonymous objects:

- First, a class is constructed, e.g. `Align`, whose constructor expects multiple arguments. In our example representing, respectively, the field width and the alignment.
- Furthermore, we define the function:

  ```
  ostream &operator<<(ostream &ostr, Align const &align)
  ```

  so we can insert an `Align` object into the ostream.

Here is an example of a little program using such a *home-made* manipulator expecting multiple arguments:

```
#include <iostream>
#include <iomanip>

class Align
{
    unsigned d_width;
    std::ios::fmtflags d_alignment;

    public:
        Align(unsigned width, std::ios::fmtflags alignment);
        std::ostream &operator()(std::ostream &ostr) const;
};

    Align::Align(unsigned width, std::ios::fmtflags alignment)
    :
        d_width(width),
        d_alignment(alignment)
    {}

    std::ostream &Align::operator()(std::ostream &ostr) const
    {
        ostr.setf(d_alignment, std::ios::adjustfield);
        return ostr << std::setw(d_width);
    }

std::ostream &operator<<(std::ostream &ostr, Align const &align)
{
    return align(ostr);
}

using namespace std;
```

```
int main()
{
    cout
        << "'" << Align(5, ios::left) << "hi" << "'"
        << "'" << Align(10, ios::right) << "there" << "'\n";
}

/*
    Generated output:

    'hi   ''     there'
*/
```

Note that in order to insert an anonymous `Align` object into the `ostream`, the `operator<<` function *must* define a `Align const &` parameter (note the `const` modifier).

10.11 The case of [io]fstream::open()

Earlier, in section 6.4.2.1, it was noted that the `[io]fstream::open` members expect an `ios::openmode` value as their final argument. E.g., to open an `fstream` object for writing you could do as follows:

```
fstream out;
out.open("/tmp/out", ios::out);
```

Combinations are also possible. To open an `fstream` object for *both* reading and writing the following stanza is often seen:

```
fstream out;
out.open("/tmp/out", ios::in | ios::out);
```

When trying to combine enum values using a 'home made' `enum` we may run into problems. Consider the following:

```
enum Permission
{
    READ =      1 << 0,
    WRITE =     1 << 1,
    EXECUTE =   1 << 2
};

void setPermission(Permission permission);

int main()
{
    setPermission(READ | WRITE);
}
```

When offering this little program to the compiler it will reply with an error message like this:

```
invalid conversion from 'int' to 'Permission'
```

The question is of course: why is it OK to combine `ios::openmode` values passing these combined values to the stream's `open` member, but not OK to combine `Permission` values.

Combining enum values using arithmetic operators results in `int`-typed values. *Conceptually* this never was our intention. Conceptually it can be considered correct to combine enum values if the resulting value conceptually makes sense as a value that is still within the original enumeration domain. Note that after adding a value `READWRITE = READ |WRITE` to the above `enum` we're still not allowed to specify `READ |WRITE` as an argument to `setPermission`.

To answer the question about combining enumeration values and yet stay within the enumeration's domain we turn to operator overloading. Up to this point operator overloading has been applied to class types. Free functions like `operator«` have been overloaded, and those overloads are conceptually within the domain of their class.

As **C++** is a stronly typed language realize that defining an `enum` is really something beyond the mere association of `int`-values with symbolic names. An enumeration type is really a type of its own, and as with any type its operators can be overloaded. When writing `READ |WRITE` the compiler will perform the default converstion from enum values to `int` values and thus will apply the operator to `int`s. It does so when it has no alternative.

But it is also possible to overload the enum type's operators. Thus we may ensure that we'll remain within the enum's domain even though the resulting value wasn't defined by the enum. The advantage of type-safety and conceptual clarity is considered to outweigh the somewhat peculiar introduction of values hitherto not defined by the enum.

Here is an example of such an overloaded operator:

```
Permission operator|(Permission left, Permission right)
{
    return static_cast<Permission>(static_cast<int>(left) | right);
}
```

Other operators can easily and analogously be constructed.

Operators like the above were defined for the `ios::openmode` enumeration type, allowing us to specify `ios::in |ios::out` as argument to `open` while specifying the corresponding parameter as `ios::openmode` as well. Clearly, operator overloading can be used in many situations, not necessarily only involving class-types.

10.12 Overloadable operators

The following operators can be overloaded:

```
+       -       *       /       %       ^       &       |
~       !       ,       =       <       >       <=      >=
++      --      <<      >>      ==      !=      &&      ||
+=      -=      *=      /=      %=      ^=      &=      |=
<<=     >>=     []      ()      ->      ->*     new     new[]
delete  delete[]
```

Several operators have *textual alternatives*:

textual alternative	operator
`and`	`&&`
`and_eq`	`&=`
`bitand`	`&`
`bitor`	`|`
`compl`	`~`
`not`	`!`
`not_eq`	`!=`
`or`	`||`
`or_eq`	`|=`
`xor`	`^`
`xor_eq`	`^=`

'Textual' alternatives of operators are also overloadable (e.g., `operator and`). However, note that textual alternatives are not *additional* operators. So, within the same context `operator&&` and `operator and` can not *both* be overloaded.

Several of these operators may only be overloaded as member functions *within* a class. This holds true for the `'='`, the `'[]'`, the `'()'` and the `'->'` operators. Consequently, it isn't possible to redefine, e.g., the assignment operator globally in such a way that it accepts a `char const *` as an `lvalue` and a `String &` as an *rvalue*. Fortunately, that isn't necessary either, as we have seen in section 10.3.

Finally, the following operators cannot be overloaded:

```
.       .*      ::      ?:      sizeof  typeid
```

Chapter 11

Static Data And Functions

In the previous chapters we provided examples of classes where each object had its own set of data members data. Each of the class's member functions could access any member of any object of its class.

In some situations it may be desirable to define *common data fields*, that may be accessed by *all* objects of the class. For example, the name of the startup directory, used by a program that recursively scans the directory tree of a disk. A second example is a variable that indicates whether some specific initialization has occurred. In that case the object that was constructed first would perform the initialization and would set the flag to 'done'.

Such situations are als encountered in **C**, where several functions need to access the same variable. A common solution in **C** is to define all these functions in one source file and to define the variable `static`: the variable name will not be visible beyond the scope of the source file. This approach is quite valid, but violates our philosophy of using only one function per source file. Another **C**-solution is to give the variable in question an unusual name, e.g., `_6uldv8`, hoping that other program parts won't use this name by accident. Neither the first, nor the second legacy **C** solution is elegant.

C++ solves the problem by defining `static members`: data and functions, common to all objects of a class and (when defined in the private section) inaccessible outside of the class. These static members are this chapter's topic.

Static members cannot be defined as virtual functions. A virtual member function is an ordinary member in that it has a `this` pointer. As static member functions have no `this` pointer, they cannot be declared virtual.

11.1 Static data

Any data member of a class can be declared `static`; be it in the `public` or `private` section of the class interface. Such a data member is created and initialized only once, in contrast to non-static data members which are created again and again for each object of the class.

Static data members are created as soon as the program starts. Even though they're created at the very beginning of a program's execution cycle they are nevertheless true members of their classes.

It is suggested to prefix the names of static member with `s_` so they may easily distinguished (in class member functions) from the class's data members (which should preferably start with `d_`).

Public static data members are global variables. They may be accessed by *all of the program's code*, simply by using their class names, the scope resolution operator and their member names. Example:

```
class Test
{
    static int s_private_int;
```

```
    public:
        static int s_public_int;
};

int main()
{
    Test::s_public_int = 145;   // OK
    Test::s_private_int = 12;   // wrong, don't touch
                                // the private parts
}
```

No executable program will result from the example. It merely illustrates the *interface*, and not the *implementation* of `static` data members, which is discussed next.

11.1.1 Private static data

To illustrate the use of a static data member which is a private variable in a class, consider the following:

```
class Directory
{
    static char s_path[];

    public:
        // constructors, destructors, etc.
};
```

The data member `s_path[]` is a private static data member. During the program's execution only *one* `Directory::s_path[]` exists, even though multiple objects of the class `Directory` may exist. This data member could be inspected or altered by the constructor, destructor or by any other member function of the class `Directory`.

Since constructors are called for each new object of a class, static data members are not *initialized* by constructors. At most they are *modified*. The reason for this is that static data members exist *before* any constructor of the class has been called. Static data members are initialized when they are defined, outside of any member function, exactly like the initialization of ordinary (non-class) global variables.

The definition and initialization of a static data member usually occurs in one of the source files of the class functions, preferably in a source file dedicated to the definition of static data members, called `data.cc`.

The data member `s_path[]`, used above, could thus be defined and initialized as follows in a file `data.cc`:

```
include "directory.ih"

char Directory::s_path[200] = "/usr/local";
```

In the class interface the static member is actually only *declared*. In its implementation (definition) its type and class name are explicitly mentioned. Note also that the size specification can be left out of the interface, as shown above. However, its size *is* (either explicitly or implicitly) required when it is defined.

Note that *any* source file could contain the definition of the static data members of a class. A separate `data.cc` source file is advised, but the source file containing, e.g., `main()` could be used as well. Of course, any source file defining static data of a class must also include the header file of that class, in order for the static data member to be known to the compiler.

A second example of a useful private static data member is given below. Assume that a class `Graphics` defines the communication of a program with a graphics-capable device (e.g., a VGA screen). The initialization of the device, which in this case would be to switch from text mode to graphics mode, is an action of the constructor and depends on a `static` flag variable `s_nobjects`. The variable `s_nobjects` simply counts the number of `Graphics` objects which are present at one time. Similarly, the destructor of the class may switch back from graphics mode to text mode when the last `Graphics` object ceases to exist. The class interface for this `Graphics` class might be:

```
class Graphics
{
    static int s_nobjects;              // counts # of objects

    public:
        Graphics();
        ~Graphics();                    // other members not shown.
    private:
        void setgraphicsmode();         // switch to graphics mode
        void settextmode();             // switch to text-mode
}
```

The purpose of the variable `s_nobjects` is to count the number of objects existing at a particular moment in time. When the first object is created, the graphics device is initialized. At the destruction of the last `Graphics` object, the switch from graphics mode to text mode is made:

```
int Graphics::s_nobjects = 0;           // the static data member

Graphics::Graphics()
{
    if (!s_nobjects++)
        setgraphicsmode();
}

Graphics::~Graphics()
{
    if (!--s_nobjects)
        settextmode();
}
```

Obviously, when the class `Graphics` would define more than one constructor, each constructor would need to increase the variable `s_nobjects` and would possibly have to initialize the graphics mode.

11.1.2 Public static data

Data members could also be declared in the public section of a class. This, however, is deprecated (as it violates the principle of data hiding). The static data member `s_path[]` (cf. section 11.1) could be declared in the public section of the class definition. This would allow all the program's code to access this variable directly:

```
int main()
{
    getcwd(Directory::s_path, 199);
}
```

A declaration is not a definition. Consequently the variable `s_path` still has to be defined. This implies that some source file still needs to contain `s_path[]` array's definition.

11.1.3 Initializing static const data

Static `const` data members may be initialized in the class interface if these data members are of integral or built-in primitive data types. So, in the following example the first three static data members can be initialized since `int` and `double` are primitive built-in types and `int` and `enum` types are integral types. The static data member `s_str` cannot be initialized in the class interface since `string` is neither a primitive built-in nor an integral data type:

```
class X
{
    public:
        enum Enum
        {
            FIRST,
        };

        static int const s_x = 34;
        static Enum const s_type = FIRST;

        static double const s_d = 1.2;
        static string const s_str = "a";    // won't compile
};
```

The compiler *may* decide to initialize static `const` data members as mere constant values, in which they don't have addresses. If the compiler does so, such static `const` data members behave as though they were values of an `enum` defined by the class. Consequently they are not variables and so it is not possible to determine their addresses. Note that trying to obtain the address of such a constant value does not create a compilation problem, but it *does* create a linking problem as the static `const` variable that is initialized as a mere constant value does not exist in addressable memory.

A statement like `int *ip = &X::s_x` may therefore *compile* correctly, but may then fail to *link*. Static variables that are explicitly defined in a source file *can* be linked correctly, though. So, in the following example the address of `X::s_x` cannot be solved by the linker, but the address of `X::s_y` *can* be determined:

```
class X
{
    public:
        static int const s_x = 34;
        static int const s_y;
};

int const X::s_y = 12;

int main()
{
    int const *ip = &X::s_x;    // compiles, but fails to link
    ip = &X::s_y;               // compiles and links correctly
}
```

11.2 Static member functions

In addition to static data members, **C++** allows us to define *static member functions*. Similar to static data that are shared by all objects of the class, static member functions also exist without any associated object of their class.

Static member functions can access all static members of their class, but *also* the members (`private` or `public`) of objects of their class *if* they are informed about the existence of these objects (as in the upcoming example). As static member functions are not associated with any object of their class they do not have a `this` pointer. In fact, a static member function is completely comparable to a global function, not associated with any class (i.e., in practice they are. See the next section (11.2.1) for a subtle note). Since static member functions do not require an associated object, static member functions declared in the public section of a class interface may be called without specifying an object of its class. The following example illustrates this characteristic of static member functions:

```
class Directory
{
    string d_currentPath;
    static char s_path[];

    public:
        static void setpath(char const *newpath);
        static void preset(Directory &dir, char const *path);
};
inline void Directory::preset(Directory &dir, char const *newpath)
{
                                                // see the text below
    dir.d_currentPath = newpath;                // 1
}

char Directory::s_path[200] = "/usr/local";     // 2

void Directory::setpath(char const *newpath)
{
    if (strlen(newpath) >= 200)
        throw "newpath too long";

    strcpy(s_path, newpath);                    // 3
}

int main()
{
    Directory dir;

    Directory::setpath("/etc");                 // 4
    dir.setpath("/etc");                        // 5

    Directory::preset(dir, "/usr/local/bin");   // 6
    dir.preset(dir, "/usr/local/bin");          // 7
}
```

- at 1 a static member function modifies a private data member of an object. However, the object whose member must be modified is given to the member function as a reference parameter.

  Note that static member functions can be defined as inline functions.

- at 2 a relatively long array is defined to be able to accomodate long paths. Alternatively, a `string` or a pointer to dynamic memory could be used.

- at 3 a (possibly longer, but not too long) new pathname is stored in the static data member `s_path[]`. Note that only static members are used.

- at 4, `setpath()` is called. It is a static member, so no object is required. But the compiler must know to which class the function belongs, so the class is mentioned using the scope resolution operator.

- at 5, the same is implemented as in 4. Here `dir` is used to tell the compiler that we're talking about a function in the `Directory` class. Static member functions *can* be called as normal member functions, but this does not imply that the static member function receives the object's address as a `this` pointer. Here the member-call syntax is used as an alternative for the classname plus scope resolution operator syntax.
- at 6, `currentPath` is altered. As in 4, the class and the scope resolution operator are used.
- at 7, the same is implemented as in 6. But here `dir` is used to tell the compiler that we're talking about a function in the `Directory` class. Here in particular note that this is *not* using `preset()` as an ordinary member function of `dir`: the function still has no `this`-pointer, so `dir` must be passed as argument to inform the static member function `preset` about the object whose `currentPath` member it should modify.

In the example only public static member functions were used. **C++** also allows the definition of private static member functions. Such functions can only be called by member functions of their class.

11.2.1 Calling conventions

As noted in the previous section, static (public) member functions are comparable to classless functions. However, formally this statement is not true, as the **C++** standard does not prescribe the same calling conventions for static member functions as for classless global functions.

In practice the calling conventions are identical, implying that the address of a static member function could be used as an argument of functions having parameters that are pointers to (global) functions.

If unpleasant surprises must be avoided at all cost, it is suggested to create global classless *wrapper functions* around static member functions that must be used as *call back* functions for other functions.

Recognizing that the traditional situations in which call back functions are used in **C** are tackled in **C++** using template algorithms (cf. chapter 19), let's assume that we have a class `Person` having data members representing the person's name, address, phone and weight. Furthermore, assume we want to sort an array of pointers to `Person` objects, by comparing the `Person` objects these pointers point to. Keeping things simple, we assume that the following public static member exists:

```
int Person::compare(Person const *const *p1, Person const *const *p2);
```

A useful characteristic of this member is that it may directly inspect the required data members of the two `Person` objects passed to the member function using double pointers.

Most compilers will allow us to pass this function's address as the address of the comparison function for the standard **C** `qsort()` function. E.g.,

```
qsort
(
    personArray, nPersons, sizeof(Person *),
    reinterpret_cast<int(*)(const void *, const void *)>(Person::compare)
);
```

However, if the compiler uses different calling conventions for static members and for classless functions, this might not work. In such a case, a classless wrapper function like the following may be used profitably:

```
int compareWrapper(void const *p1, void const *p2)
{
    return
        Person::compare
        (
```

```
            reinterpret_cast<Person const *const *>(p1),
            reinterpret_cast<Person const *const *>(p2)
        );
}
```

resulting in the following call of the `qsort()` function:

```
qsort(personArray, nPersons, sizeof(Person *), compareWrapper);
```

Note:

- The wrapper function takes care of any mismatch in the calling conventions of static member functions and classless functions;
- The wrapper function handles the required type casts;
- The wrapper function might perform small additional services (like dereferencing pointers if the static member function expects references to `Person` objects rather than double pointers);
- As an aside: in **C++** programs functions like `qsort()`, requiring the specification of call back functions are seldom used. Instead using existing generic template algorithms is preferred (cf. chapter 19).

Chapter 12

Abstract Containers

C++ offers several predefined datatypes, all part of the Standard Template Library, which can be used to implement solutions to frequently occurring problems. The datatypes discussed in this chapter are all *containers*: you can put stuff inside them, and you can retrieve the stored information from them.

The interesting part is that the kind of data that can be stored inside these containers has been left unspecified at the time the containers were constructed. That's why they are spoken of as *abstract* containers.

Abstract containers rely heavily on *templates*, covered in chapter 20 and beyonf. To use abstract containers, only a minimal grasp of the template concept is required. In **C++** a template is in fact a recipe for constructing a function or a complete class. The recipe tries to abstract the functionality of the class or function as much as possible from the data on which the class or function operates. As the data types on which the templates operate were not known when the template was implemented, the datatypes are either inferred from the context in which a function template is used, or they are mentioned explicitly when a class template is used (the term that's used here is *instantiated*). In situations where the types are explicitly mentioned, the *angle bracket notation* is used to indicate which data types are required. For example, below (in section 12.2) we'll encounter the `pair` container, which requires the explicit mentioning of two data types. Here is a `pair` object containing both an `int` and a `string`:

```
pair<int, string> myPair;
```

The object `myPair` is defined as an object holding both an `int` and a `string`.

The angle bracket notation is used intensively in the upcoming discussion of abstract containers. Actually, understanding this part of templates is the only real requirement for using abstract containers. Now that we've introduced this notation, we can postpone the more thorough discussion of templates to chapter 20, and concentrate on their use in this chapter.

Most of the abstract containers are *sequential* containers: they contain data that can be stored and retrieved in some sequential way. Examples are the `vector`, implementing an extendable array; the `list`, implementing a datastructure that allows for the easy insertion or deletion of data the `queue`, also called a *FIFO* (first in, first out) structure, in which the first element that is entered will be the first element that will be retrieved; and the `stack`, which is a *first in, last out* (FILO or LIFO) structure.

In addition to sequential containers several special containers are available. The `pair` is a basic container in which a pair of values (of types that are left open for further specification) can be stored, like two strings, two ints, a string and a double, etc.. Pairs are often used to return data elements that naturally come in pairs. For example, the `map` is an abstract container storing keys and their associated values. Elements of these maps are returned as `pairs`.

A variant of the `pair` is the `complex` container, implementing operations that are defined on *complex numbers*.

All abstract containers described in this chapter as well as the `string` and stream datatypes (cf. chap-

ters 5 and 6) are part of the Standard Template Library.

All containers support the following operators:

- The overloaded assignment operator, so we can assign two containers of the same types to each other.
- Tests for equality: `==` and `!=` The equality operator applied to two containers returns `true` if the two containers have the same number of elements, which are pairwise equal according to the equality operator of the contained data type. The inequality operator does the opposite.
- Ordering operators: `<`, `<=`, `>` and `>=`. The `<` operator returns `true` if each element in the left-hand side container is less than each corresponding element in the right-hand side container. Additional elements in either the left-hand side container or the right-hand side container are ignored.

```
container left;
container right;

left = {0, 2, 4};
right = {1, 3};             // left < right

right = {1, 3, 6, 1, 2};    // left < right
```

Note that before a user-defined type (usually a `class`-type) can be stored in a container, the user-defined type should at least support:

- A default-value (e.g., a default constructor)
- The equality operator (==)
- The less-than operator (<)

With the advent of the C++0x standard sequential containers can also be initialized using *initializer lists*.

Most containers (exceptions are the stack (section 12.3.10), priority_queue (section 12.3.4), and queue (section 12.3.3) containers) support members to determine their maximum sizes (through their member max_size()).

Closely linked to the standard template library are the *generic algorithms*. These algorithms may be used to perform frequently occurring tasks or more complex tasks than is possible with the containers themselves, like counting, filling, merging, filtering etc.. An overview of generic algorithms and their applications is given in chapter 19. Generic algorithms usually rely on the availability of *iterators*, representing begin and end-points for processing data stored inside containers. The abstract containers usually support constructors and members expecting iterators, and they often have members returning iterators (comparable to the `string::begin()` and `string::end()` members). In this chapter the iterator concept is not further investigated. Refer to chapter 18 for this.

The url `http://www.sgi.com/Technology/STL` is worth visiting as it offers more exensive coverage of abstract containers and the standard template library than can be provided by the **C++** annotations.

Containers often collect data during their lifetimes. When a container goes out of scope, its destructor tries to destroy its data elements. This only succeeds if the data elements themselves are stored inside the container. If the data elements of containers are pointers to dynamically allocated memory then the data pointed to by these pointers will not be destroyed, resulting in a memory leak. A consequence of this scheme is that the data stored in a container should often be considered the 'property' of the container: the container should be able to destroy its data elements when the container's destructor is called. So, normally containers should not contain pointers to data. Also, a container should not be required to contain `const` data, as `const` data prevent the use of many of the container's members, like the assignment operator.

12.1 Notations used in this chapter

In this chapter about containers, the following notational convention is used:

- Containers live in the standard namespace. In code examples this will be clearly visible, but in the text `std::` is usually omitted.
- A container without angle brackets represents any container of that type. Mentally add the required type in angle bracket notation. E.g., `pair` may represent `pair<string, int>`.
- The notation `Type` represents the generic type. `Type` could be `int`, `string`, etc.
- Identifiers `object` and `container` represent objects of the container type under discussion.
- The identifier `value` represents a value of the type that is stored in the container.
- Simple, one-letter identifiers, like `n` represent unsigned values.
- Longer identifiers represent iterators. Examples are `pos`, `from`, `beyond`

Some containers, e.g., the `map` container, contain pairs of values, usually called 'keys' and 'values'. For such containers the following notational convention is used in addition:

- The identifier `key` indicates a value of the used key-type
- The identifier `keyvalue` indicates a value of the '`value_type`' used with the particular container.

12.2 The 'pair' container

The `pair` container is a rather basic container. It is used to store two elements, called `first` and `second`, and that's about it. Before using `pair` containers the header file `<utility>` must have been included.

The `pair`'s data types are specified when the `pair` object is defined (or declared) using the template's angle bracket notation (cf. chapter 20). Examples:

```
pair<string, string> piper("PA28", "PH-ANI");
pair<string, string> cessna("C172", "PH-ANG");
```

here, the variables `piper` and `cessna` are defined as `pair` variables containing two `strings`. Both strings can be retrieved using the `first` and `second` fields of the `pair` type:

```
cout << piper.first << endl <<      // shows 'PA28'
        cessna.second << endl;      // shows 'PH-ANG'
```

The `first` and `second` members can also be used to reassign values:

```
cessna.first = "C152";
cessna.second = "PH-ANW";
```

If a `pair` object must be completely reassigned, an *anonymous* pair object can be used as the right-hand operand of the assignment. An anonymous variable defines a temporary variable (which receives no name) solely for the purpose of (re)assigning another variable of the same type. Its generic form is

```
type(initializer list)
```

Note that when a `pair` object is used the type specification is not completed by just mentioning the containername `pair`. It also requires the specification of the data types which are stored within the pair. For this the (template) angle bracket notation is used again. E.g., the reassignment of the `cessna` pair variable could have been accomplished as follows:

```
cessna = pair<string, string>("C152", "PH-ANW");
```

In cases like these, the `type` specification can become quite elaborate, which has caused a revival of interest in the possibilities offered by the `typedef` keyword. If many `pair<type1, type2>` clauses are used in a source, the typing effort may be reduced and readability might be improved by first defining a name for the clause, and then using the defined name later. E.g.,

```
typedef pair<string, string> pairStrStr;

cessna = pairStrStr("C152", "PH-ANW");
```

Apart from this (and the basic set of operations (assignment and comparisons)) the `pair` offers no further functionality. It is, however, a basic ingredient of the upcoming abstract containers `map`, `multimap` and `hash_map`.

The C++0x standard offers a *generalized pair* container: the *tuple*, covered in section 21.5.4.

12.3 Sequential Containers

12.3.1 The 'vector' container

The `vector` class implements an expandable array. Before using the `vector` container the `<vector>` header file must have been included.

The following constructors, operators, and member functions are available:

- Constructors:
  - A `vector` may be constructed empty:

    ```
    vector<string> object;
    ```

    Note the specification of the data type to be stored in the `vector`: the data type is given between angle brackets, just after the 'vector' container name. This is common practice with containers.
  - A vector may be initialized to a certain number of elements. One of the nicer characteristics of vectors (and other containers) is that it initializes its data elements to the data type's default value. The data type's *default constructor* is used for this initialization. With non-class data types the value 0 is used. So, for the `int` vector we know its initial values are zero. With the advent of the C++0x standard (starting `g++` release 4.4) it has also become possible to use initializer lists. Some examples:

    ```
    vector<string> object(5, string("Hello")); // initialize to 5 Hello's,
    vector<string> container(10);              // and to 10 empty strings
    vector<string> names = {"george", "frank", "tony", "karel"}; // g++ 4.4
    ```
  - A vector may be initialized using iterators. To initialize a vector with elements 5 until 10 (including the last one) of an existing `vector<string>` the following construction may be used:

    ```
    extern vector<string> container;
    vector<string> object(&container[5], &container[11]);
    ```

Note here that the last element pointed to by the second iterator (`&container[11]`) is *not* stored in `object`. This is a simple example of the use of *iterators*, in which the range of values that is used starts at the first value, and includes all elements up to but not including the element to which the second iterator refers. The standard notation for this is `[begin, end)`.

- A vector may be initialized using a copy constructor:

```
extern vector<string> container;
vector<string> object(container);
```

- In addition to the standard operators for containers, the `vector` supports the index operator, which may be used to retrieve or reassign individual elements of the vector. Note that the elements which are indexed must exist. For example, having defined an empty vector a statement like `ivect[0] = 18` produces an error, as the vector is empty. So, the vector is *not* automatically expanded, and it *does* respect its array bounds. In this case the vector should be resized first, or `ivect.push_back(18)` should be used (see below).

- The `vector` class offers the following member functions:
  - `Type &back()`:

    this member returns a reference to the last element in the vector. It is the responsibility of the programmer to use the member only if the vector is not empty.
  - `vector::iterator begin()`:

    this member returns an iterator pointing to the first element in the vector, returning `end` if the vector is empty.
  - `size_t capacity()`:

    Number of elements for which memory has been allocated. It returns at least the value returned by `size`
  - `void clear()`:

    this member erases all the vector's elements.
  - `bool empty()`

    this member returns `true` if the vector contains no elements.
  - `vector::iterator end()`:

    this member returns an iterator pointing beyond the last element in the vector.
  - `vector::iterator erase()`:

    this member can be used to erase a specific range of elements in the vector:
    - `erase(pos)` erases the element pointed to by the iterator `pos`. The iterator `++pos` is returned.
    - `erase(first, beyond)` erases elements indicated by the iterator range `[first, beyond)`, returning `beyond`.
  - `Type &front()`:

    this member returns a reference to the first element in the vector. It is the responsibility of the programmer to use the member only if the vector is not empty.
  - `... insert()`:

    elements may be inserted starting at a certain position. The return value depends on the version of `insert()` that is called:
    - `vector::iterator insert(pos)` inserts a default value of type `Type` at `pos`, `pos` is returned.
    - `vector::iterator insert(pos, value)` inserts `value` at `pos`, `pos` is returned.
    - `void insert(pos, first, beyond)` inserts the elements in the iterator range `[first, beyond)`.
    - `void insert(pos, n, value)` inserts `n` elements having value `value` at position `pos`.

- `void pop_back()`:

  this member removes the last element from the vector. With an empty vector nothing happens.

- `void push_back(value)`:

  this member adds `value` to the end of the vector.

- `void reserve(size_t request)`:

  if `request` is less than or equal to `capacity`, this call has no effect. Otherwise, it is a request to allocate additional memory. If the call is successful, then `capacity` returns a value of at least `request`. Otherwise, `capacity` is unchanged. In either case, `size`'s return value won't change, until a function like `resize` is called, actually changing the number of accessible elements.

- `void resize()`:

  this member can be used to alter the number of elements that are currently stored in the vector:

  * `resize(n, value)` may be used to resize the vector to a size of `n`. `Value` is optional. If the vector is expanded and `value` is not provided, the additional elements are initialized to the default value of the used data type, otherwise `value` is used to initialize extra elements.

- `vector::reverse_iterator rbegin()`:

  this member returns an iterator pointing to the last element in the vector.

- `vector::reverse_iterator rend()`:

  this member returns an iterator pointing before the first element in the vector.

- `size_t size()`

  this member returns the number of elements in the vector.

- `void swap()`

  this member can be used to swap two vectors using identical data types. Example:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v1(7);
    vector<int> v2(10);

    v1.swap(v2);
    cout << v1.size() << " " << v2.size() << '\n';
}
/*
    Produced output:
10 7
*/
```

12.3.2 The 'list' container

The `list` container implements a list data structure. Before using a `list` container the header file `<list>` must have been included.

The organization of a `list` is shown in figure 12.1. Igure 12.1 shows that a list consists of separate list-elements, connected by pointers. The list can be traversed in two directions: starting at *Front* the list may be traversed from left to right, until the 0-pointer is reached at the end of the rightmost list-element. The list can also be traversed from right to left: starting at *Back*, the list is traversed from right to left, until eventually the 0-pointer emanating from the leftmost list-element is reached.

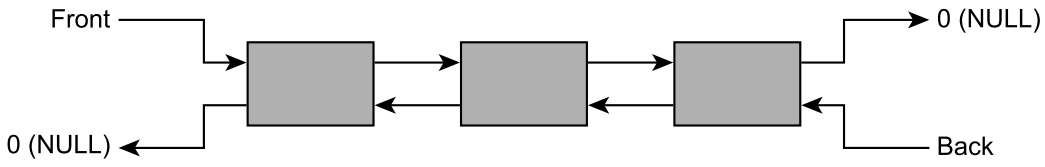

Figure 12.1: A list data-structure

As a subtlety note that the representation given in figure 12.1 is not necessarily used in actual implementations of the list. For example, consider the following little program:

```
int main()
{
    list<int> l;
    cout << "size: " << l.size() << ", first element: " <<
            l.front() << '\n';
}
```

When this program is run it might actually produce the output:

```
size: 0, first element: 0
```

Its front element can even be assigned a value. In this case the implementor has choosen to provide the list with a hidden element. The list actually is a *circular* list, where the hidden element serves as terminating element, replacing the 0-pointers in figure 12.1. As noted, this is a subtlety, which doesn't affect the conceptual notion of a list as a data structure ending in 0-pointers. Note also that it is well known that various implementations of list-structures are possible (cf. Aho, A.V., Hopcroft J.E. and Ullman, J.D., (1983) *Data Structures and Algorithms* (Addison-Wesley)).

Both lists and vectors are often appropriate data structures in situations where an unknown number of data elements must be stored. However, there are some rules of thumb to follow when selecting the appropriate data structure.

- When most accesses are random, a `vector` is the preferred data structure. Example: in a program counting character frequencies in a textfile, a `vector<int> frequencies(256)` is the datastructure of choice, as the values of the received characters can be used as indices into the `frequencies` vector.
- The previous example illustrates a second rule of thumb, also favoring the `vector`: if the number of elements is known in advance (and does not notably change during the lifetime of the program), the vector is also preferred over the list.
- In cases where insertions or deletions prevail and the data structure is large the list is generally preferred.

At present lists aren't as useful anymore as they used to be (when computers were much slower and more memory-constrained). Except maybe for some rare cases, a `vector` should be the preferred container; even when implementing algorithms traditionally using lists.

Other considerations related to the choice between lists and vectors should also be given some thought. Although it is true that the vector is able to grow dynamically, the dynamic growth requires data-copying. Clearly, copying a million large data structures takes a considerable amount of time, even on fast computers. On the other hand, inserting a large number of elements in a list doesn't require us to copy non-involved data. Inserting a new element in a list merely requires us to juggle some pointers. In figure 12.2 this is shown: a new element is inserted between the second and third element, creating a new list of four elements. Removing an element from a list is also fairly easy. Starting again from the

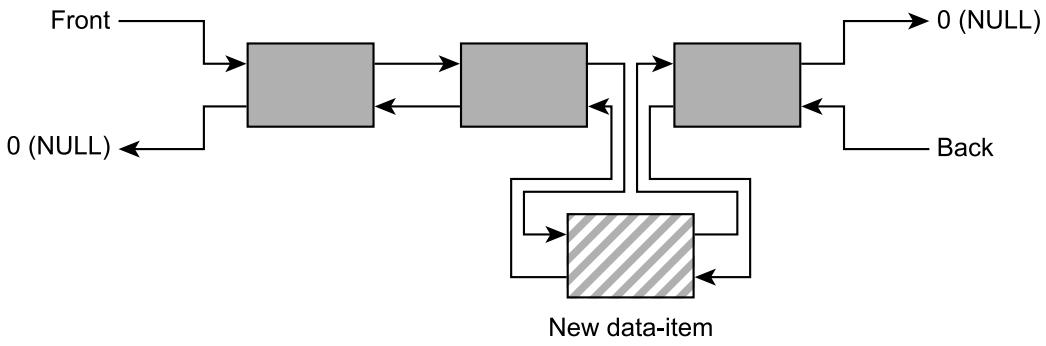

Figure 12.2: Adding a new element to a list

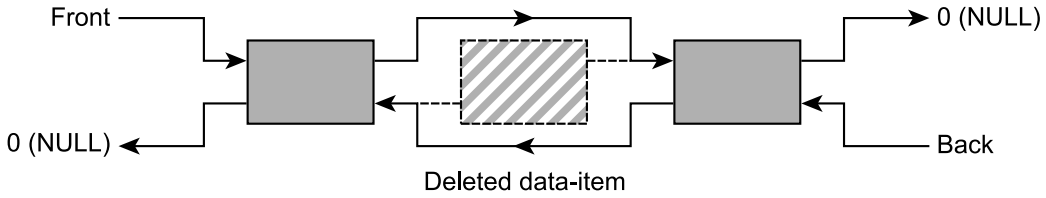

Figure 12.3: Removing an element from a list

situation shown in figure 12.1, figure 12.3 shows what happens if element two is removed from our list. Again: only pointers need to be juggled. In this case it's even simpler than adding an element: only two pointers need to be rerouted. To summarize the comparison between lists and vectors: it's probably best to conclude that there is no clear-cut answer to the question what data structure to prefer. There are rules of thumb, which may be adhered to. But if worse comes to worst, a profiler may be required to find out what's best.

The `list` container offers the following constructors, operators, and member functions are available:

- Constructors:
  - A `list` may be constructed empty:

    ```
    list<string> object;
    ```

    As with the `vector`, it is an error to refer to an element of an empty list.
  - A list may be initialized to a certain number of elements. By default, if the initialization value is not explicitly mentioned, the default value or default constructor for the actual data type is used. For example:

    ```
    list<string> object(5, string("Hello")); // initialize to 5 Hello's
    list<string> container(10);              // and to 10 empty strings
    ```
  - A list may be initialized using a two iterators. To initialize a list with elements 5 until 10 (including the last one) of a `vector<string>` the following construction may be used:

    ```
    extern vector<string> container;
    list<string> object(&container[5], &container[11]);
    ```
  - A list may be initialized using a copy constructor:

    ```
    extern list<string> container;
    list<string> object(container);
    ```
- The `list` does not offer specialized operators, apart from the standard operators for containers.

- The following member functions are available:
  - `Type &back()`:

    this member returns a reference to the last element in the list. It is the responsibility of the programmer to use this member only if the list is not empty.
  - `list::iterator begin()`:

    this member returns an iterator pointing to the first element in the list, returning `end` if the list is empty.
  - `void clear()`:

    this member erases all elements from the list.
  - `bool empty()`:

    this member returns `true` if the list contains no elements.
  - `list::iterator end()`:

    this member returns an iterator pointing beyond the last element in the list.
  - `list::iterator erase()`:

    this member can be used to erase a specific range of elements in the list:
    - `erase(pos)` erases the element pointed to by `pos`. The iterator `++pos` is returned.
    - `erase(first, beyond)` erases elements indicated by the iterator range `[first, beyond)`. `Beyond` is returned.
  - `Type &front()`:

    this member returns a reference to the first element in the list. It is the responsibility of the programmer to use this member only if the list is not empty.
  - `... insert()`:

    this member can be used to insert elements into the list. The return value depends on the version of `insert` that is called:
    - `list::iterator insert(pos)` inserts a default value of type `Type` at `pos`, `pos` is returned.
    - `list::iterator insert(pos, value)` inserts `value` at `pos`, `pos` is returned.
    - `void insert(pos, first, beyond)` inserts the elements in the iterator range `[first, beyond)`.
    - `void insert(pos, n, value)` inserts `n` elements having value `value` at position `pos`.
  - `void list<Type>::merge(list<Type> other)`:

    this member function assumes that the current and other lists are sorted (see below, the member `sort`), and will, based on that assumption, insert the elements of `other` into the current list in such a way that the modified list remains sorted. If both list are not sorted, the resulting list will be ordered 'as much as possible', given the initial ordering of the elements in the two lists. `list<Type>::merge` uses `Type::operator<` to sort the data in the list, which operator must therefore be available. The next example illustrates the use of the `merge` member: the list '`object`' is not sorted, so the resulting list is ordered 'as much as possible'.

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

void showlist(list<string> &target)
{
    for
    (
        list<string>::iterator from = target.begin();
        from != target.end();
```

```
            ++from
        )
            cout << *from << " ";

        cout << '\n';
    }

    int main()
    {
        list<string> first;
        list<string> second;

        first.push_back(string("alpha"));
        first.push_back(string("bravo"));
        first.push_back(string("golf"));
        first.push_back(string("quebec"));

        second.push_back(string("oscar"));
        second.push_back(string("mike"));
        second.push_back(string("november"));
        second.push_back(string("zulu"));

        first.merge(second);
        showlist(first);
    }
```

A subtlety is that `merge` doesn't alter the list if the list itself is used as argument: `object.merge(object)` won't change the list '`object`'.

- `void pop_back()`:

  this member removes the last element from the list. With an empty list nothing happens.

- `void pop_front()`:

  this member removes the first element from the list. With an empty list nothing happens.

- `void push_back(value)`:

  this member adds `value` to the end of the list.

- `void push_front(value)`:

  this member adds `value` before the first element of the list.

- `void resize()`:

  this member can be used to alter the number of elements that are currently stored in the list:

  * `resize(n, value)` may be used to resize the list to a size of `n`. `Value` is optional. If the list is expanded and `value` is not provided, the extra elements are initialized to the default value of the used data type, otherwise `value` is used to initialize extra elements.

- `list::reverse_iterator rbegin()`:

  this member returns an iterator pointing to the last element in the list.

- `void remove(value)`:

  this member removes all occurrences of `value` from the list. In the following example, the two strings '`Hello`' are removed from the list `object`:

```
    #include <iostream>
    #include <string>
    #include <list>
    using namespace std;

    int main()
```

```
        {
            list<string> object;

            object.push_back(string("Hello"));
            object.push_back(string("World"));
            object.push_back(string("Hello"));
            object.push_back(string("World"));

            object.remove(string("Hello"));

            while (object.size())
            {
                cout << object.front() << '\n';
                object.pop_front();
            }
        }
        /*
                Generated output:
            World
            World
        */
```

- `list::reverse_iterator rend()`:

  this member returns an iterator pointing before the first element in the list.

- `size_t size()`:

  this member returns the number of elements in the list.

- `void reverse()`:

  this member reverses the order of the elements in the list. The element `back` will become `front` and *vice versa*.

- `void sort()`:

  this member will sort the list. Once the list has been sorted, An example of its use is given at the description of the `unique` member function below. `list<Type>::sort` uses `Type::operator<` to sort the data in the list, which operator must therefore be available.

- `void splice(pos, object)`:

  this member function transfers the contents of `object` to the current list, starting the insertion at the iterator position `pos` of the object using the `splice` member. Following `splice`, `object` is empty. For example:

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
    list<string> object;

    object.push_front(string("Hello"));
    object.push_back(string("World"));

    list<string> argument(object);

    object.splice(++object.begin(), argument);

    cout << "Object contains " << object.size() << " elements, " <<
            "Argument contains " << argument.size() <<
            " elements,\n";
```

```
    while (object.size())
    {
        cout << object.front() << '\n';
        object.pop_front();
    }
}
```

Alternatively, `argument` may be followed by an iterator of `argument`, indicating the first element of `argument` that should be spliced, or by two iterators `begin` and `end` defining the iterator-range `[begin, end)` on `argument` that should be spliced into `object`.

- `void swap()`:

  this member can be used to swap two lists using identical data types.

- `void unique()`:

  operating on a sorted list, this member function will remove all consecutively identical elements from the list. `list<Type>::unique` uses `Type::operator==` to identify identical data elements, which operator must therefore be available. Here's an example removing all multiply occurring words from the list:

```
#include <iostream>
#include <string>
#include <list>
using namespace std;
                            // see the merge() example
void showlist(list<string> &target);
void showlist(list<string> &target)
{
    for
    (
        list<string>::iterator from = target.begin();
        from != target.end();
        ++from
    )
        cout << *from << " ";

    cout << '\n';
}


int main()
{
    string
        array[] =
        {
            "charley",
            "alpha",
            "bravo",
            "alpha"
        };

    list<string>
        target
        (
            array, array + sizeof(array)
            / sizeof(string)
        );

    cout << "Initially we have:\n";
    showlist(target);
```

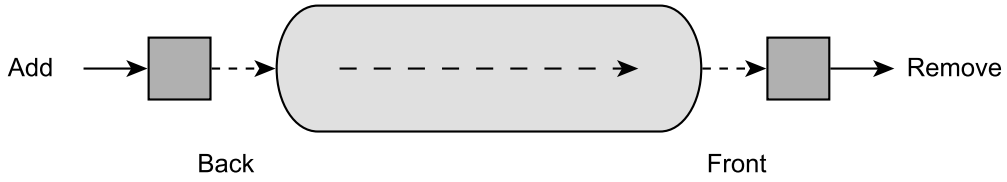

Figure 12.4: A queue data-structure

```
    target.sort();
    cout << "After sort() we have:\n";
    showlist(target);

    target.unique();
    cout << "After unique() we have:\n";
    showlist(target);
}
/*
    Generated output:

    Initially we have:
    charley alpha bravo alpha
    After sort() we have:
    alpha alpha bravo charley
    After unique() we have:
    alpha bravo charley
*/
```

12.3.3 The 'queue' container

The `queue` class implements a queue data structure. Before using a `queue` container the header file `<queue>` must have been included.

A queue is depicted in figure 12.4. In figure 12.4 it is shown that a queue has one point (the *back*) where items can be added to the queue, and one point (the *front*) where items can be removed (read) from the queue. A `queue` is therefore also called a *FIFO* data structure, for *first in, first out*. It is most often used in situations where events should be handled in the same order as they are generated.

The following constructors, operators, and member functions are available for the `queue` container:

- Constructors:
  - A `queue` may be constructed empty:

    ```
    queue<string> object;
    ```

    As with the `vector`, it is an error to refer to an element of an empty queue.
  - A queue may be initialized using a copy constructor:

    ```
    extern queue<string> container;
    queue<string> object(container);
    ```
- The `queue` container only supports the basic container operators.
- The following member functions are available for queues:
  - `Type &back()`:

    this member returns a reference to the last element in the queue. It is the responsibility of the programmer to use the member only if the queue is not empty.

- `bool empty()`:

  this member returns `true` if the queue contains no elements.

- `Type &front()`:

  this member returns a reference to the first element in the queue. It is the responsibility of the programmer to use the member only if the queue is not empty.

- `void push(value)`:

  this member adds `value` to the back of the queue.

- `void pop()`:

  this member removes the element at the front of the queue. Note that the element is *not* returned by this member. Nothing happens if the member is called for an empty queue. One might wonder why `pop` returns `void`, instead of a value of type `Type` (cf. `front`). One reason is found in the principles of good software design: functions should perform one task. Combining the removal and return of the removed element breaks this principle. Moreover, when this principle is abandoned `pop`'s implementation will always be flawed. Consider the prototypical implementation of a `pop` member that is supposed to return the just popped value:

  ```
  Type queue::pop()
  {
      Type ret(top());
      erase_top();
      return ret;
  }
  ```

  The venom, as usual, is in the tail: since `queue` has no control over `Type`'s behavior the final statement (`return ret`) might throw. By that time the queue's topmost element has already been removed from the queue and so it is lost. Thus, a `Type` returning `pop` member cannot offer the *strong guarantee* and consequently `pop` should not return the former `top` element. Because of all this, we must first use `front`, and then `pop` to obtain and remove the queue's front element.

- `size_t size()`:

  this member returns the number of elements in the queue.

Note that the queue does not support iterators or a subscript operator. The only elements that can be accessed are its front and back element. A queue can be emptied by:

- repeatedly removing its front element;
- assigning an empty queue using the same data type to it;
- having its destructor called.

12.3.4 The 'priority_queue' container

The `priority_queue` class implements a priority queue data structure. Before using a `priority_queue` container the `<queue>` header file must have been included.

A priority queue is identical to a `queue`, but allows the entry of data elements according to *priority rules*. A real-life priority queue is found, e.g., at airport check-in terminals. At a terminal the passengers normally stand in line to wait for their turn to check in, but late passengers are usually allowed to jump the queue: they receive a higher priority than other passengers.

The priority queue uses `operator<` of the data type stored in the priority ueue to decide about the priority of the data elements. The *smaller* the value, the *lower* the priority. So, the priority queue *could* be used to sort values while they arrive. A simple example of such a priority queue application is the following program: it reads words from `cin` and writes a sorted list of words to `cout`:

```
#include <iostream>
```

```
#include <string>
#include <queue>
using namespace std;

int main()
{
    priority_queue<string> q;
    string word;

    while (cin >> word)
        q.push(word);

    while (q.size())
    {
        cout << q.top() << '\n';
        q.pop();
    }
}
```

Unfortunately, the words are listed in reversed order: because of the underlying <-operator the words appearing later in the ASCII-sequence appear first in the priority queue. A solution to that problem is to define a *wrapper class* around the `string` datatype, reversing `string`'s `operator<`. Here is the modified program:

```
#include <iostream>
#include <string>
#include <queue>

class Text
{
    std::string d_s;

    public:
        Text(std::string const &str)
        :
            d_s(str)
        {}
        operator std::string const &() const
        {
            return d_s;
        }
        bool operator<(Text const &right) const
        {
            return d_s > right.d_s;
        }
};

using namespace std;

int main()
{
    priority_queue<Text> q;
    string word;

    while (cin >> word)
        q.push(word);

    while (q.size())
    {
```

```
        word = q.top();
        cout << word << endl;
        q.pop();
    }
}
```

Other possibilities to achieve the same exist. One would be to store the contents of the priority queue in, e.g., a vector, from which the elements can be read in reversed order.

The following constructors, operators, and member functions are available for the `priority_queue` container:

- Constructors:
  - A `priority_queue` may be constructed empty:

    ```
    priority_queue<string> object;
    ```

    As with the `vector`, it is an error to refer to an element of an empty priority queue.
  - A priority queue may be initialized using a copy constructor:

    ```
    extern priority_queue<string> container;
    priority_queue<string> object(container);
    ```

- The `priority_queue` only supports the basic operators of containers.
- The following member functions are available for priority queues:
  - `bool empty()`:

    this member returns `true` if the priority queue contains no elements.
  - `void push(value)`:

    this member inserts `value` at the appropriate position in the priority queue.
  - `void pop()`:

    this member removes the element at the top of the priority queue. Note that the element is *not* returned by this member. Nothing happens if this member is called for and empty priority queue. See section 12.3.3 for a discussion about the reason why `pop` has return type `void`.
  - `size_t size()`:

    this member returns the number of elements in the priority queue.
  - `Type &top()`:

    this member returns a reference to the first element of the priority queue. It is the responsibility of the programmer to use the member only if the priority queue is not empty.

Note that the priority queue does not support iterators or a subscript operator. The only element that can be accessed is its top element. A priority queue can be emptied by:

- repeatedly removing its top element;
- assigning an empty queue using the same data type to it;
- having its destructor called.

12.3.5 The ‘deque’ container

The `deque` (pronounce: ‘deck’) class implements a doubly ended queue data structure (deque). Before using a `deque` container the header file `<deque>` must have been included.

A `deque` is comparable to a queue, but it allows for reading and writing at both ends. Actually, the `deque` data type supports a lot more functionality than the `queue`, as will be clear from the following overview of available member functions. A `deque` is a combination of a `vector` and two queues, operating at both ends of the vector. In situations where random insertions and the addition and/or removal of elements at one or both sides of the vector occurs frequently using a `deque` should be considered.

The following constructors, operators, and member functions are available for deques:

- Constructors:
  - A `deque` may be constructed empty:

    ```
    deque<string>
        object;
    ```

    As with the `vector`, it is an error to refer to an element of an empty deque.
  - A deque may be initialized to a certain number of elements. By default, if the initialization value is not explicitly mentioned, the default value or default constructor for the actual data type is used. For example:

    ```
    deque<string> object(5, string("Hello")), // initialize to 5 Hello's
    deque<string> container(10);              // and to 10 empty strings
    ```
  - A deque may be initialized using two iterators. To initialize a deque with elements 5 until 10 (including the last one) of a `vector<string>` the following construction may be used:

    ```
    extern vector<string> container;
    deque<string> object(&container[5], &container[11]);
    ```
  - A deque may be initialized using a copy constructor:

    ```
    extern deque<string> container;
    deque<string> object(container);
    ```
- In addition to the standard operators for containers, the deque supports the index operator, which may be used to retrieve or reassign random elements of the deque. Note that the indexed elements must exist.
- The following member functions are available for deques:
  - `Type &back()`:

    this member returns a reference to the last element in the deque. It is the responsibility of the programmer to use the member only if the deque is not empty.
  - `deque::iterator begin()`:

    this member returns an iterator pointing to the first element in the deque.
  - `void clear()`:

    this member erases all elements in the deque.
  - `bool empty()`:

    this member returns `true` if the deque contains no elements.
  - `deque::iterator end()`:

    this member returns an iterator pointing beyond the last element in the deque.
  - `deque::iterator erase()`:

    the member can be used to erase a specific range of elements in the deque:
    - `erase(pos)` erases the element pointed to by `pos`. The iterator `++pos` is returned.

    * `erase(first, beyond)` erases elements indicated by the iterator range `[first, beyond)`. `Beyond` is returned.

- `Type &front()`:
  this member returns a reference to the first element in the deque. It is the responsibility of the programmer to use the member only if the deque is not empty.
- `... insert()`:
  this member can be used to insert elements starting at a certain position. The return value depends on the version of `insert` that is called:
  * `deque::iterator insert(pos)` inserts a default value of type `Type` at `pos`, `pos` is returned.
  * `deque::iterator insert(pos, value)` inserts `value` at `pos`, `pos` is returned.
  * `void insert(pos, first, beyond)` inserts the elements in the iterator range `[first, beyond)`.
  * `void insert(pos, n, value)` inserts `n` elements having value `value` starting at iterator position `pos`.
- `void pop_back()`:
  this member removes the last element from the deque. With an empty deque nothing happens.
- `void pop_front()`:
  this member removes the first element from the deque. With an empty deque nothing happens.
- `void push_back(value)`:
  this member adds `value` to the end of the deque.
- `void push_front(value)`:
  this member adds `value` before the first element of the deque.
- `void resize()`:
  this member can be used to alter the number of elements that are currently stored in the deque:
  * `resize(n, value)` may be used to resize the deque to a size of `n`. `Value` is optional. If the deque is expanded and `value` is not provided, the additional elements are initialized to the default value of the used data type, otherwise `value` is used to initialize extra elements.
- `deque::reverse_iterator rbegin()`:
  this member returns an iterator pointing to the last element in the deque.
- `deque::reverse_iterator rend()`:
  this member returns an iterator pointing before the first element in the deque.
- `size_t size()`:
  this member returns the number of elements in the deque.
- `void swap(argument)`:
  this member can be used to swap two deques using identical data types.

12.3.6 The 'map' container

The `map` class offers a (sorted) associative array. Before using a `map` container the `<map>` header file must have been included.

A `map` is filled with *key*/*value* pairs, which may be of any container-accepted type. Since types are associated with both the key and the value, we must specify *two types* in the angle bracket notation, comparable to the specification we've seen with the `pair` container (cf. section 12.2). The first type represents the key's type, the second type represents the value's type. For example, a `map` in which the key is a `string` and the value is a `double` can be defined as follows:

```
map<string, double> object;
```

The *key* is used to access its associated information. That information is called the *value*. For example, a phone book uses the names of people as the key, and uses the telephone number and maybe other information (e.g., the zip-code, the address, the profession) as value. Since a `map` sorts its keys, the `key`'s `operator<` must be defined, and it must be sensible to use it. For example, it is generally a bad idea to use pointers for keys, as sorting pointers is something different than sorting the values pointed at by those pointers.

The two fundamental operations on maps are the storage of *Key*/*Value* combinations, and the retrieval of values, given their keys. The index operator using a key as the index, can be used for both. If the index operator is used as *lvalue*, insertion will be performed. If it is used as *rvalue*, the key's associated value is retrieved. Each key can be stored only once in a `map`. If the same key is entered again, the new value replaces the formerly stored value, which is lost.

A specific key/value combination can implicitly or explicitly be inserted into a `map`. If explicit insertion is required, the key/value combination must be constructed first. For this, every `map` defines a `value_type` which may be used to create values that can be stored in the `map`. For example, a value for a `map<string, int>` can be constructed as follows:

```
map<string, int>::value_type siValue("Hello", 1);
```

The `value_type` is associated with the `map<string, int>`: the type of the key is `string`, the type of the value is `int`. Anonymous `value_type` objects are also often used. E.g.,

```
map<string, int>::value_type("Hello", 1);
```

Instead of using the line `map<string, int>::value_type(...)` over and over again, a `typedef` is frequently used to reduce typing and to improve readability:

```
typedef map<string, int>::value_type StringIntValue
```

Using this typedef, values for the `map<string, int>` may now be constructed using:

```
StringIntValue("Hello", 1);
```

Alternatively, `pairs` may be used to represent key/value combinations used by maps:

```
pair<string, int>("Hello", 1);
```

The following constructors, operators, and member functions are available for the `map` container:

- Constructors:
  - A `map` may be constructed empty:

    ```
    map<string, int> object;
    ```

    Note that the values stored in maps may be containers themselves. For example, the following defines a `map` in which the value is a `pair`: a container nested under another container:

    ```
    map<string, pair<string, string>> object;
    ```

    Note the use of the two consecutive closing angle brackets. Before the advent of the C++0x standard consecutive closing brackets in container type specifications (and generally: in the context of template type specifications) resulted in a compilation error, as the immediate concatenation of the two closing angle brackets would be interpreted by the compiler as a right shift operator (`operator>>`), which is not what we want here. In compilers supporting the C++0x standard this construction is accepted. Compilers not yet implementing this feature require a separating blank between two consecutive closing angle brackets.

– A map may be initialized using two iterators. The iterators may either point to `value_type` values for the map to be constructed, or to plain `pair` objects. If pairs are used, their `first` element represents the type of the keys, and their `second` element represents the type of the values. Example:

```
pair<string, int> pa[] =
{
    pair<string,int>("one", 1),
    pair<string,int>("two", 2),
    pair<string,int>("three", 3),
};

map<string, int> object(&pa[0], &pa[3]);
```

In this example, `map<string, int>::value_type` could have been written instead of `pair<string, int>` as well.

If `begin` represents the first iterator that us used to construct a map and if `end` represents the second iterator, `[begin, end)` will be used to initialize the map. Maybe contrary to intuition, the `map` constructor will only enter *new* keys. If the last element of `pa` would have been `"one", 3`, only *two* elements would have entered the `map`: `"one", 1` and `"two", 2`. The value `"one", 3` would silently have been ignored.

The `map` receives its own copies of the data to which the iterators point as illustrated by the following example:

```
#include <iostream>
#include <map>
using namespace std;

class MyClass
{
    public:
        MyClass()
        {
            cout << "MyClass constructor\n";
        }
        MyClass(const MyClass &other)
        {
            cout << "MyClass copy constructor\n";
        }
        ~MyClass()
        {
            cout << "MyClass destructor\n";
        }
};

int main()
{
    pair<string, MyClass> pairs[] =
    {
        pair<string, MyClass>("one", MyClass())
    };
    cout << "pairs constructed\n";

    map<string, MyClass> mapsm(&pairs[0], &pairs[1]);
    cout << "mapsm constructed\n";
}
/*
    Generated output:
MyClass constructor
MyClass copy constructor
MyClass destructor
```

```
pairs constructed
MyClass copy constructor
MyClass copy constructor
MyClass destructor
mapsm constructed
MyClass destructor
MyClass destructor
*/
```

When tracing the output of this program, we see that, first, the constructor of a `MyClass` object is called to initialize the anonymous element of the array `pairs`. This object is then copied into the first element of the array `pairs` by the copy constructor. Next, the original element is not required anymore and is destroyed. At that point the array `pairs` has been constructed. Thereupon, the `map` constructs a temporary `pair` object, which is used to construct the map element. Having constructed the map element, the temporary `pair` object is destroyed. Eventually, when the program terminates, the `pair` element stored in the `map` is destroyed too.

- A map may be initialized using a copy constructor:

```
extern map<string, int> container;
map<string, int> object(container);
```

- Apart from the standard operators for containers, the `map` supports the index operator, which may be used to retrieve or reassign individual elements of the map. Here, the argument of the index operator is a key. If the provided key is not available in the `map`, a new data element is automatically added to the `map` using the default value or default constructor to initialize the value part of the new element. This default value is returned if the index operator is used as an rvalue.

  When initializing a new or reassigning another element of the map, the type of the right-hand side of the assignment operator must be equal to (or promotable to) the type of the map's value part. E.g., to add or change the value of element `"two"` in a map, the following statement can be used:

```
mapsm["two"] = MyClass();
```

- The `map` class has the following member functions:
  - `map::iterator begin()`:
    this member returns an iterator pointing to the first element of the map.
  - `void clear()`:
    this member erases all elements from the map.
  - `size_t count(key)`:
    this member returns 1 if the provided key is available in the `map`, otherwise 0 is returned.
  - `bool empty()`:
    this member returns `true` if the map contains no elements.
  - `map::iterator end()`:
    this member returns an iterator pointing beyond the last element of the map.
  - `pair<map::iterator, map::iterator> equal_range(key)`:
    this member returns a pair of iterators, being respectively the return values of the member functions `lower_bound` and `upper_bound`, introduced below. An example illustrating these member functions is given at the discussion of the member function `upper_bound`.
  - `... erase()`:
    this member can be used to erase a specific element or range of elements from the map:

  * `bool erase(key)` erases the element having the given `key` from the `map`. `True` is returned if the value was removed, `false` if the map did not contain an element using the given `key`.
  * `void erase(pos)` erases the element pointed to by the iterator `pos`.
  * `void erase(first, beyond)` erases all elements indicated by the iterator range `[first, beyond)`.

– `map::iterator find(key)`:

this member returns an iterator to the element having the given key. If the element isn't available, `end` is returned. The following example illustrates the use of the `find` member function:

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<string, int> object;

    object["one"] = 1;

    map<string, int>::iterator it = object.find("one");

    cout << "'one' " <<
            (it == object.end() ? "not " : "") << "found\n";

    it = object.find("three");

    cout << "'three' " <<
            (it == object.end() ? "not " : "") << "found\n";
}
/*
    Generated output:
'one' found
'three' not found
*/
```

– `... insert()`:

this member can be used to insert elements into the map. It will, however, not replace the values associated with already existing keys by new values. Its return value depends on the version of `insert` that is called:

  * `pair<map::iterator, bool> insert(keyvalue)` inserts a new `value_type` into the map. The return value is a `pair<map::iterator, bool>`. If the returned `bool` field is `true`, `keyvalue` was inserted into the map. The value `false` indicates that the key that was specified in `keyvalue` was already available in the map, and so `keyvalue` was not inserted into the map. In both cases the `map::iterator` field points to the data element having the `key` that was specified in `keyvalue`. The use of this variant of `insert` is illustrated by the following example:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("one", 10),
```

```
            pair<string,int>("two", 20),
            pair<string,int>("three", 30),
        };
        map<string, int> object(&pa[0], &pa[3]);

                // {four, 40} and 'true' is returned
        pair<map<string, int>::iterator, bool>
            ret = object.insert
                    (
                        map<string, int>::value_type
                        ("four", 40)
                    );

        cout << boolalpha;

        cout << ret.first->first << " " <<
            ret.first->second << " " <<
            ret.second << " " << object["four"] << endl;

                // {four, 40} and 'false' is returned
        ret = object.insert
                    (
                        map<string, int>::value_type
                        ("four", 0)
                    );

        cout << ret.first->first << " " <<
            ret.first->second << " " <<
            ret.second << " " << object["four"] << endl;
    }
    /*
        Generated output:

        four 40 true 40
        four 40 false 40
    */
```

Note the somewhat peculiar constructions like

```
cout << ret.first->first << " " << ret.first->second << ...
```

Note that 'ret' is equal to the pair returned by the insert member function. Its 'first' field is an iterator into the map<string, int>, so it can be considered a pointer to a map<string, int>::value_type. These value types themselves are pairs too, having 'first' and 'second' fields. Consequently, 'ret.first->first' is the *key* of the map value (a string), and 'ret.first->second' is the *value* (an int).

* map::iterator insert(pos, keyvalue). This way a map::value_type may also be inserted into the map. pos is ignored, and an iterator to the inserted element is returned.
* void insert(first, beyond) inserts the (map::value_type) elements pointed to by the iterator range [first, beyond).

– map::iterator lower_bound(key):

  this member returns an iterator pointing to the first keyvalue element of which the key is at least equal to the specified key. If no such element exists, the function returns end.

– map::reverse_iterator rbegin():

  this member returns an iterator pointing to the last element of the map.

– map::reverse_iterator rend():

  this member returns an iterator pointing before the first element of the map.

- `size_t size()`:

  this member returns the number of elements in the map.

- `void swap(argument)`:

  this member can be used to swap two maps using identical key/value types.

- `map::iterator upper_bound(key)`:

  this member returns an iterator pointing to the first `keyvalue` element having a `key` exceeding the specified `key`. If no such element exists, the function returns `end`. The following example illustrates the member functions `equal_range`, `lower_bound` and `upper_bound`:

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("one", 10),
        pair<string,int>("two", 20),
        pair<string,int>("three", 30),
    };
    map<string, int> object(&pa[0], &pa[3]);
    map<string, int>::iterator it;

    if ((it = object.lower_bound("tw")) != object.end())
        cout << "lower-bound 'tw' is available, it is: " <<
                it->first << '\n';

    if (object.lower_bound("twoo") == object.end())
        cout << "lower-bound 'twoo' not available" << '\n';

    cout << "lower-bound two: " <<
            object.lower_bound("two")->first <<
            " is available\n";

    if ((it = object.upper_bound("tw")) != object.end())
        cout << "upper-bound 'tw' is available, it is: " <<
                it->first << '\n';

    if (object.upper_bound("twoo") == object.end())
        cout << "upper-bound 'twoo' not available" << '\n';

    if (object.upper_bound("two") == object.end())
        cout << "upper-bound 'two' not available" << '\n';

    pair
    <
        map<string, int>::iterator,
        map<string, int>::iterator
    >
        p = object.equal_range("two");

    cout << "equal range: 'first' points to " <<
                p.first->first << ", 'second' is " <<
        (
            p.second == object.end() ?
                "not available"
            :
```

```
                        p.second->first
                ) <<
                '\n';
        }
        /*
            Generated output:

                lower-bound 'tw' is available, it is: two
                lower-bound 'twoo' not available
                lower-bound two: two is available
                upper-bound 'tw' is available, it is: two
                upper-bound 'twoo' not available
                upper-bound 'two' not available
                equal range: 'first' points to two, 'second' is not available
        */
```

As mentioned at the beginning of this section, the `map` represents a sorted associative array. In a `map` the keys are sorted. If an application must visit all elements in a map the `begin` and `end` iterators must be used. The following example shows how to make a simple table listing all keys and values found in a map:

```
#include <iostream>
#include <iomanip>
#include <map>

using namespace std;

int main()
{
    pair<string, int>
        pa[] =
        {
            pair<string,int>("one", 10),
            pair<string,int>("two", 20),
            pair<string,int>("three", 30),
        };
    map<string, int>
        object(&pa[0], &pa[3]);

    for
    (
        map<string, int>::iterator it = object.begin();
            it != object.end();
                ++it
    )
        cout << setw(5) << it->first.c_str() <<
                setw(5) << it->second << '\n';
}
/*
    Generated output:
  one   10
three   30
  two   20
*/
```

12.3.7 The 'multimap' container

Like the `map`, the `multimap` class implements a (sorted) associative array. Before using a `multimap` container the header file `<map>` must have been included.

The main difference between the `map` and the `multimap` is that the multimap supports multiple values associated with the same key, whereas the map contains single-valued keys. Note that the multimap also accepts multiple identical values associated with identical keys.

The `map` and the `multimap` have the same set of member functions, with the exception of the index operator which is not supported with the multimap. This is understandable: if multiple entries of the same key are allowed, which of the possible values should be returned for `object[key]`?

Refer to section 12.3.6 for an overview of the `multimap` member functions. Some member functions, however, deserve additional attention when used in the context of the `multimap` container. These members are discussed below.

- `size_t map::count(key)`:

  this member returns the number of entries in the multimap associated with the given `key`.

- `... erase()`:

  this member can be used to erase elements from the map:

  - `size_t erase(key)` erases all elements having the given `key`. The number of erased elements is returned.
  - `void erase(pos)` erases the single element pointed to by `pos`. Other elements possibly having the same keys are not erased.
  - `void erase(first, beyond)` erases all elements indicated by the iterator range `[first, beyond)`.

- `pair<multimap::iterator, multimap::iterator> equal_range(key)`:

  this member function returns a pair of iterators, being respectively the return values of `lower_bound` and `upper_bound`, introduced below. The function provides a simple means to determine all elements in the `multimap` that have the same `keys`. An example illustrating the use of these member functions is given at the end of this section.

- `multimap::iterator find(key)`:

  this member returns an iterator pointing to the first value whose key is `key`. If the element isn't available, `end` is returned. The iterator could be incremented to visit all elements having the same `key` until it is either `end`, or the iterator's `first` member is not equal to `key` anymore.

- `multimap::iterator insert()`:

  this member function normally succeeds, and so a *multimap::iterator* is returned, instead of a `pair<multimap::iterator, bool>` as returned with the `map` container. The returned iterator points to the newly added element.

Although the functions `lower_bound` and `upper_bound` act identically in the `map` and `multimap` containers, their operation in a `multimap` deserves some additional attention. The next example illustrates `lower_bound`, `upper_bound` and `equal_range` applied to a `multimap`:

```
#include <iostream>
#include <map>
using namespace std;
```

```
int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("alpha", 1),
        pair<string,int>("bravo", 2),
        pair<string,int>("charley", 3),
        pair<string,int>("bravo", 6),   // unordered 'bravo' values
        pair<string,int>("delta", 5),
        pair<string,int>("bravo", 4),
    };
    multimap<string, int> object(&pa[0], &pa[6]);

    typedef multimap<string, int>::iterator msiIterator;

    msiIterator it = object.lower_bound("brava");

    cout << "Lower bound for 'brava': " <<
            it->first << ", " << it->second << endl;

    it = object.upper_bound("bravu");

    cout << "Upper bound for 'bravu': " <<
            it->first << ", " << it->second << endl;

    pair<msiIterator, msiIterator>
        itPair = object.equal_range("bravo");

    cout << "Equal range for 'bravo':\n";
    for (it = itPair.first; it != itPair.second; ++it)
        cout << it->first << ", " << it->second << endl;
    cout << "Upper bound: " << it->first << ", " << it->second << endl;

    cout << "Equal range for 'brav':\n";
    itPair = object.equal_range("brav");
    for (it = itPair.first; it != itPair.second; ++it)
        cout << it->first << ", " << it->second << endl;
    cout << "Upper bound: " << it->first << ", " << it->second << endl;
}
/*
    Generated output:

    Lower bound for 'brava': bravo, 2
    Upper bound for 'bravu': charley, 3
    Equal range for 'bravo':
    bravo, 2
    bravo, 6
    bravo, 4
    Upper bound: charley, 3
    Equal range for 'brav':
    Upper bound: bravo, 2
*/
```

In particular note the following characteristics:

- `lower_bound` and `upper_bound` produce the same result for non-existing keys: they both return the first element having a key that exceeds the provided key.
- Although the keys are ordered in the `multimap`, the values for equal keys are not ordered: they

are retrieved in the order in which they were enterd.

12.3.8 The 'set' container

The `set` class implements a sorted collection of values. Before using `set` containers the `<set>` header file must have been included.

A set contains unique values (of a container-acceptable type). Each value is stored only once.

A specific value can be explicitly created: Every `set` defines a `value_type` which may be used to create values that can be stored in the `set`. For example, a value for a `set<string>` can be constructed as follows:

```
set<string>::value_type setValue("Hello");
```

The `value_type` is associated with the `set<string>`. Anonymous `value_type` objects are also often used. E.g.,

```
set<string>::value_type("Hello");
```

Instead of using the line `set<string>::value_type(...)` over and over again, a `typedef` is often used to reduce typing and to improve readability:

```
typedef set<string>::value_type StringSetValue
```

Using this typedef, values for the `set<string>` may be constructed as follows:

```
StringSetValue("Hello");
```

Alternatively, values of the set's type may be used immediately. In that case the value of type `Type` is implicitly converted to a `set<Type>::value_type`.

The following constructors, operators, and member functions are available for the `set` container:

- Constructors:
  - A `set` may be constructed empty:
    ```
    set<int> object;
    ```
  - A set may be initialized using two iterators. For example:
    ```
    int intarr[] = {1, 2, 3, 4, 5};

    set<int> object(&intarr[0], &intarr[5]);
    ```
    Note that all values in the set must be different: it is not possible to store the same value repeatedly when the set is constructed. If the same value occurs repeatedly, only the first instance of the value will be entered, the other values will be silently ignored.

    Like the map, the `set` receives its own copy of the data it contains.
- A set may be initialized using a copy constructor:
  ```
  extern set<string> container;
  set<string> object(container);
  ```
- The `set` container only supports the standard set of operators that are available for containers.

- The `set` class has the following member functions:
  - `set::iterator begin()`:

    this member returns an iterator pointing to the first element of the set. If the set is empty `end` is returned.
  - `void clear()`:

    this member erases all elements from the set.
  - `size_t count(key)`:

    this member returns 1 if the provided key is available in the `set`, otherwise 0 is returned.
  - `bool empty()`:

    this member returns `true` if the set contains no elements.
  - `set::iterator end()`:

    this member returns an iterator pointing beyond the last element of the set.
  - `pair<set::iterator, set::iterator> equal_range(key)`:

    this member returns a pair of iterators, being respectively the return values of the member functions `lower_bound` and `upper_bound`, introduced below.
  - `... erase()`:

    this member can be used to erase a specific element or range of elements from the set:
    - `bool erase(value)` erases the element having the given `value` from the `set`. `True` is returned if the value was removed, `false` if the set did not contain an element '`value`'.
    - `void erase(pos)` erases the element pointed to by the iterator `pos`.
    - `void erase(first, beyond)` erases all elements indicated by the iterator range `[first, beyond)`.
  - `set::iterator find(value)`:

    this member returns an iterator to the element having the given value. If the element isn't available, `end` is returned.
  - `... insert()`:

    this member can be used to insert elements into the `set`. If the element already exists, the existing element is left untouched and the element to be inserted is ignored. The return value depends on the version of `insert` that is called:
    - `pair<set::iterator, bool> insert(keyvalue)` inserts a new `set::value_type` into the set. The return value is a `pair<set::iterator, bool>`. If the returned `bool` field is `true`, `value` was inserted into the set. The value `false` indicates that the value that was specified was already available in the set, and so the provided `value` was not inserted into the set. In both cases the `set::iterator` field points to the data element in the `set` having the specified `value`.
    - `set::iterator insert(pos, keyvalue)`. This way a `set::value_type` may also be inserted into the set. `pos` is ignored, and an iterator to the inserted element is returned.
    - `void insert(first, beyond)` inserts the (`set::value_type`) elements pointed to by the iterator range `[first, beyond)` into the set.
  - `set::iterator lower_bound(key)`:

    this member returns an iterator pointing to the first `keyvalue` element of which the `key` is at least equal to the specified `key`. If no such element exists, the function returns `end`.
  - `set::reverse_iterator rbegin()`:

    this member returns an iterator pointing to the last element of the set.
  - `set::reverse_iterator rend`:

    this member returns an iterator pointing before the first element of the set.

- `size_t size()`:

  this member returns the number of elements in the set.

- `void swap(argument)`:

  this member can be used to swap two sets (`argument` being the second set) that use identical data types.

- `set::iterator upper_bound(key)`:

  this member returns an iterator pointing to the first `keyvalue` element having a `key` exceeding the specified `key`. If no such element exists, the function returns `end`.

12.3.9 The 'multiset' container

Like the `set`, the `multiset` class implements a sorted collection of values. Before using `multiset` containers the header file `<set>` must have been included.

The main difference between the `set` and the `multiset` is that the multiset supports multiple entries of the same value, whereas the set contains unique values.

The `set` and the `multiset` have the same set of member functions. Refer to section 12.3.8 for an overview of the `multiset` member functions. Some member functions, however, behave slightly different than their counterparts of the `set` container. Those members are mentioned here.

- `size_t count(value)`:

  this member returns the number of entries in the multiset associated with the given `value`.

- `... erase()`:

  this member can be used to erase elements from the set:

  - `size_t erase(value)` erases all elements having the given `value`. The number of erased elements is returned.
  - `void erase(pos)` erases the element pointed to by the iterator `pos`. Other elements possibly having the same values are not erased.
  - `void erase(first, beyond)` erases all elements indicated by the iterator range `[first, beyond)`.

- `pair<multiset::iterator, multiset::iterator> equal_range(value)`:

  this member function returns a pair of iterators, being respectively the return values of `lower_bound` and `upper_bound`, introduced below. The function provides a simple means to determine all elements in the `multiset` that have the same `values`.

- `multiset::iterator find(value)`:

  this member returns an iterator pointing to the first element having the specified value. If the element isn't available, `end` is returned. The iterator could be incremented to visit all elements having the given `value` until it is either `end`, or the iterator doesn't point to '`value`' anymore.

- `... insert()`:

  this member function normally succeeds and returns a *multiset::iterator* rather than a `pair<multiset::iterator, bool>` as returned with the `set` container. The returned iterator points to the newly added element.

Although the functions lower_bound and upper_bound act identically in the set and multiset containers, their operation in a multiset deserves some additional attention. With a multiset container lower_bound and upper_bound produce the same result for non-existing keys: they both return the first element having a key exceeding the provided key.

Here is an example showing the use of various member functions of a multiset:

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    string
        sa[] =
        {
            "alpha",
            "echo",
            "hotel",
            "mike",
            "romeo"
        };

    multiset<string>
        object(&sa[0], &sa[5]);

    object.insert("echo");
    object.insert("echo");

    multiset<string>::iterator
        it = object.find("echo");

    for (; it != object.end(); ++it)
        cout << *it << " ";
    cout << '\n';

    cout << "Multiset::equal_range(\"ech\")\n";
    pair
    <
        multiset<string>::iterator,
        multiset<string>::iterator
    >
        itpair = object.equal_range("ech");

    if (itpair.first != object.end())
        cout << "lower_bound() points at " << *itpair.first << '\n';
    for (; itpair.first != itpair.second; ++itpair.first)
        cout << *itpair.first << " ";

    cout << '\n' <<
            object.count("ech") << " occurrences of 'ech'" << '\n';

    cout << "Multiset::equal_range(\"echo\")\n";
    itpair = object.equal_range("echo");

    for (; itpair.first != itpair.second; ++itpair.first)
        cout << *itpair.first << " ";
```

```
        cout << '\n' <<
                object.count("echo") << " occurrences of 'echo'" << '\n';

        cout << "Multiset::equal_range(\"echoo\")\n";
        itpair = object.equal_range("echoo");

        for (; itpair.first != itpair.second; ++itpair.first)
            cout << *itpair.first << " ";

        cout << '\n' <<
                object.count("echoo") << " occurrences of 'echoo'" << '\n';
    }
    /*
        Generated output:

        echo echo echo hotel mike romeo
        Multiset::equal_range("ech")
        lower_bound() points at echo

        0 occurrences of 'ech'
        Multiset::equal_range("echo")
        echo echo echo
        3 occurrences of 'echo'
        Multiset::equal_range("echoo")

        0 occurrences of 'echoo'
    */
```

12.3.10 The 'stack' container

The `stack` class implements a stack data structure. Before using `stack` containers the header file `<stack>` must have been included.

A stack is also called a first in, last out (FILO or LIFO) data structure as the first item to enter the stack is the last item to leave. A stack is an extremely useful data structure in situations where data must temporarily remain available. For example, programs maintain a stack to store local variables of functions: the lifetime of these variables is determined by the time these functions are active, contrary to global (or static local) variables, which live for as long as the program itself lives. Another example is found in calculators using the *Reverse Polish Notation* (RPN), in which the operands of operators are kept in a stack, whereas operators pop their operands off the stack and push the results of their work back onto the stack.

As an example of the use of a stack, consider figure 12.5, in which the contents of the stack is shown while the expression `(3 + 4) * 2` is evaluated. In the RPN this expression becomes `3 4 + 2 *`, and figure 12.5 shows the stack contents after each *token* (i.e., the operands and the operators) is read from the input. Notice that each operand is indeed pushed on the stack, while each operator changes the contents of the stack. The expression is evaluated in five steps. The caret between the tokens in the expressions shown on the first line of figure 12.5 shows what token has just been read. The next line shows the actual stack-contents, and the final line shows the steps for referential purposes. Note that at step 2, two numbers have been pushed on the stack. The first number (3) is now at the bottom of the stack. Next, in step 3, the + operator is read. The operator pops two operands (so that the stack is empty at that moment), calculates their sum, and pushes the resulting value (7) on the stack. Then, in step 4, the number 2 is read, which is dutifully pushed on the stack again. Finally, in step 5 the final operator `*` is read, which pops the values 2 and 7 from the stack, computes their product, and pushes the result back on the stack. This result (14) could then be popped to be displayed on some medium.

From figure 12.5 we see that a stack has one location (the *top*) where items can be pushed onto and popped off the stack. This top element is the stack's only immediately visible element. It may be

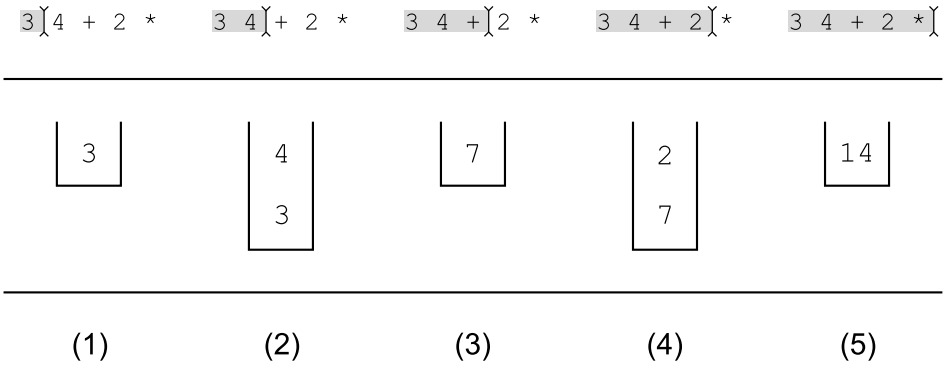

Figure 12.5: The contents of a stack while evaluating `3 4 + 2 *`

accessed and modified directly.

Bearing this model of the stack in mind, let's see what we formally can do with the `stack` container. For the `stack`, the following constructors, operators, and member functions are available:

- Constructors:
  - A `stack` may be constructed empty:
    ```
    stack<string> object;
    ```
  - A stack may be initialized using a copy constructor:
    ```
    extern stack<string> container;
    stack<string> object(container);
    ```
- Only the basic set of container operators are supported by the `stack`
- The following member functions are available for stacks:
  - `bool empty()`:
    this member returns `true` if the stack contains no elements.
  - `void push(value)`:
    this member places `value` at the top of the stack, hiding the other elements from view.
  - `void pop()`:
    this member removes the element at the top of the stack. Note that the popped element is *not* returned by this member. Nothing happens if `pop` is used with an empty stack. See section 12.3.3 for a discussion about the reason why `pop` has return type `void`.
  - `size_t size()`:
    this member returns the number of elements in the stack.
  - `Type &top()`:
    this member returns a reference to the stack's top (and only visible) element. It is the responsibility of the programmer to use this member only if the stack is not empty.

The stack does not support iterators or a subscript operator. The only elements that can be accessed is its top element. To empty a stack:

- repeatedly remove its front element;
- assign an empty stack to it;
- have its destructor called (e.g., by ending its lifetime).

12.3.11 Hash Tables (C++0x)

The C++0x standard officially adds hash tables to the language.

As discussed, the map is a sorted data structure. The keys in maps are sorted using the `operator<` of the key's data type. Generally, this is not the fastest way to either store or retrieve data. The main benefit of sorting is that a listing of sorted keys appeals more to humans than an unsorted list. However, a by far faster method to store and retrieve data is to use *hashing*.

Hashing uses a function (called the *hash function*) to compute an (unsigned) number from the key, which number is thereupon used as an index in the table in which the keys are stored. Retrieval of a key is as simple as computing the hash value of the provided key, and looking in the table at the computed index location: if the key is present, it is stored in the table, and its value can be returned. If it's not present, the key is not stored.

Collisions occur when a computed index position is already occupied by another element. For these situations the abstract containers have solutions available. A simple solution, adopted by the C++0x standard is to use *linear chaining* which uses a linked list to store colliding table elements in.

In the C++0x standard the term *unordered* is used rather than *hash* to avoid name collisions with hash tables developed before the advent of the C++0x standard. Except where `unordered` is required as part of a type name, the term *hash* will be used here as it is the term commonly encountered.

Four forms of `unordered` data structures are supported: `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`.

Below the `unordered_map` container is discussed. The other containers using hashing also use hashing but provide functionality corresponding to, respectively, the `multimap, set` and `multiset`.

Concentrating on the `unordered_map`, its constructor needs a key type, a value type, an object computing a hash value for the key, and an object comparing two keys for equality. Predefined hash functions are available for `std::string` keys, and for all standard scalar numeric types (`char, short, int` etc.). If another data type is used, a hash function object and an equality function object must be made available (see also section 10.10). Examples follow below.

The class implementing the hash function could be called `hash`. Its function call operator (`operator()`) returns the (`size_t`) hash value of the key that it received as its argument.

A *generic algorithm* (see chapter 19) exists performing tests of equality (i.e., `equal_to`). These tests can be used if the key's data type supports the equality operator. Alternatively, an overloaded `operator==` or specialized function object could be constructed returning `true` if two keys are equal and `false` otherwise. Examples follow.

The `unordered_map` class implements an associative array in which the elements are stored according to some hashing scheme. Before using `unordered_map` containers the header file `<unordered_map>` must have been included.

Constructors, operators and member functions available for the `map` are also available for the `unordered_map`. The `map` and `unordered_map` support the same set of operators and member functions. However, the *efficiency* of a `unordered_map` in terms of speed should greatly exceed the efficiency of the `map`. Comparable conclusions may be drawn for the `unordered_set`, `unordered_multimap` and the `unordered_multiset`.

Compared to the `map` container, the `unordered_map` has an additional constructor:

```
unordered_map<...> hash(n);
```

where `n` is a `size_t` value. It is used to construct a `unordered_map` consisting of an initial number of at least `n` empty slots to put key/value combinations in. This number is automatically extended when needed.

The hashed key type is almost always text. So, a `unordered_map` in which the key's data type is a `std::string` occurs most often. Note that although a `char *` is allowed as key type this is almost

always a bad idea since two `char *` variables pointing to equal **C**-strings stored at different locations will be considered different keys.

The following program defines a unordered_map containing the names of the months of the year and the number of days these months (usually) have. Then, using the subscript operator the days in several months are displayed. The equality operator used the generic algorithm `equal_to<string>`, which is the default fourth argument of the `unordered_map` constructor:

```
#include <unordered_map>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    unordered_map<string, int> months;

    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    months["may"] = 31;
    months["june"] = 30;
    months["july"] = 31;
    months["august"] = 31;
    months["september"] = 30;
    months["october"] = 31;
    months["november"] = 30;
    months["december"] = 31;

    cout << "september -> " << months["september"] << '\n' <<
            "april     -> " << months["april"] << '\n' <<
            "june      -> " << months["june"] << '\n' <<
            "november  -> " << months["november"] << '\n';
}
/*
    Generated output:
september -> 30
april     -> 30
june      -> 30
november  -> 30
*/
```

A comparable example, showing the use of explicitly defined hash and equality functions and key-type `char const *`:

```
#include <unordered_map>
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

struct EqualCp
{
    bool operator()(char const *l, char const *r) const
    {
        return strcmp(l, r) == 0;
    }
};
```

```
struct HashCp
{
    size_t operator()(char const *str) const
    {
        return hash<std::string const &>()(str);
    }
};
int main()
{
    unordered_map<char const *, int, HashCp, EqualCp> months;

    months["april"] = 30;
    months["november"] = 31;

    string apr("april");    // different pointers, same string

    cout << "april     -> " << months["april"] << '\n' <<
            "april     -> " << months[apr.c_str()] << '\n';
}
```

The `unordered_multimap`, `unordered_set` and `unordered_multiset` containers are used analogously. For these containers the `equal` and `hash` classes must also be defined. The `unordered_multimap` also requires the `unordered_map` header file.

Before using `unordered_set` and `unordered_multiset` containers the header file `<unordered_set>` must have been included.

12.3.12 Regular Expressions (C++0x, ?)

The C++0x standard adds handling of regular expressions to the language. Regular expressions were already available in **C++** via its **C** heritage as **C** has always offered functions like **regcomp**(3) and **regexec**(3),, which are used, e.g. by the `Pattern` class of the `Bobcat library`[1].

Regular expressions are extensively documented elsewhere (e.g., **regex**(7), Friedl, J.E.F `Mastering Regular Expressions`[2], O'Reilly) and the reader is referred to these sources for a refresher on the topic of regular expressions.

The C++0x standard adds native object based support for regular expressions by defining several new classes and other facilities. Currently, however, regular expressions are not yet supported by the `g++` library and therefore in this section only the basic building blocks the C++0x standard offers to handle regular expressions are mentioned. Once regular expressions actually become available this section will be updated to cover the actually available features.

Eventually, regular expressions will be represented by objects of the class `regex`. Once a `regex` regular expression object has been defined its member `regex_search` can be called to process its regular expression. This function expects arguments representing, respectively, the text against which the regular expression will be matched; an object of the class `cmatch` representing the results of the matching operation and an object of the class `regex` representing the used regular expression. Furthermore, a member `regex_replace` will be available to perform textual replacements based on regular expressions.

Before using regular expressions as offered by the C++ standard the header file `<regex>` must have been included.

Regular expressions using the `regex` class are currently not yet available in the `g++` library.

[1] http://bobcat.sourceforge.net
[2] http://oreilly.com/catalog/

12.4 The 'complex' container

The `complex` container defines the standard operations that can be performed on complex numbers. The complex number's real and imaginary types are specified as the container's data type. Examples:

```
complex<double>
complex<int>
complex<float>
```

Note that the real and imaginary parts of complex numbers have the same datatypes.

Before using `complex` containers the header file `<complex>` must have been included.

When initializing (or assigning) a complex object, the imaginary part may be omitted from the initialization or assignment resulting in its value being 0 (zero). By default, both parts are zero.

Below it is silently assumed that the used `complex` type is `complex<double>`. Given this assumption, complex numbers may be initialized as follows:

- `target`: A default initialization: real and imaginary parts are 0.
- `target(1)`: The real part is 1, imaginary part is 0
- `target(0, 3.5)`: The real part is 0, imaginary part is 3.5
- `target(source)`: `target` is initialized with the values of `source`.

Anonymous complex values may also be used. In the next example two anonymous complex values are pushed on a stack of complex numbers, to be popped again thereafter:

```
#include <iostream>
#include <complex>
#include <stack>

using namespace std;

int main()
{
    stack<complex<double>>
        cstack;

    cstack.push(complex<double>(3.14, 2.71));
    cstack.push(complex<double>(-3.14, -2.71));

    while (cstack.size())
    {
        cout << cstack.top().real() << ", " <<
                cstack.top().imag() << "i" << '\n';
        cstack.pop();
    }
}
/*
    Generated output:
-3.14, -2.71i
3.14, 2.71i
*/
```

The following member functions and operators are defined for complex numbers (below, `value` may be either a primitve scalar type or a `complex` object):

- Apart from the standard container operators, the following operators are supported from the `complex` container.
  - `complex operator+(value)`:

    this member returns the sum of the current `complex` container and `value`.
  - `complex operator-(value)`:

    this member returns the difference between the current `complex` container and `value`.
  - `complex operator*(value)`:

    this member returns the product of the current `complex` container and `value`.
  - `complex operator/(value)`:

    this member returns the quotient of the current `complex` container and `value`.
  - `complex operator+=(value)`:

    this member adds `value` to the current `complex` container, returning the new value.
  - `complex operator-=(value)`:

    this member subtracts `value` from the current `complex` container, returning the new value.
  - `complex operator*=(value)`:

    this member multiplies the current `complex` container by `value`, returning the new value
  - `complex operator/=(value)`:

    this member divides the current `complex` container by `value`, returning the new value.
- `Type real()`:

  this member returns the real part of a complex number.
- `Type imag()`:

  this member returns the imaginary part of a complex number.
- Several mathematical functions are available for the `complex` container, such as `abs`, `arg`, `conj`, `cos`, `cosh`, `exp`, `log`, `norm`, `polar`, `pow`, `sin`, `sinh` and `sqrt`. All these functions are free functions, not member functions, accepting complex numbers as their arguments. For example,

```
abs(complex<double>(3, -5));
pow(target, complex<int>(2, 3));
```

- Complex numbers may be extracted from `istream` objects and inserted into `ostream` objects. The insertion results in an ordered pair `(x, y)`, in which `x` represents the real part and `y` the imaginary part of the complex number. The same form may also be used when extracting a complex number from an `istream` object. However, simpler forms are also allowed. E.g., when extracting `1.2345` the imaginary part will be set to 0.

Chapter 13

Inheritance

When programming in **C**, programming problems are commonly approached using a top-down structured approach: functions and actions of the program are defined in terms of sub-functions, which again are defined in sub-sub-functions, etc.. This yields a hierarchy of code: `main` at the top, followed by a level of functions which are called from `main`, etc..

In **C++** the relationship between code and data is also frequently defined in terms of dependencies among *classes*. This looks like *composition* (see section 7.2), where objects of a class contain objects of another class as their data. But the relation described here is of a different kind: a class can be *defined* in terms of an older, pre-existing, class. This produces a new class having all the functionality of the older class, and additionally defining its own specific functionality. Instead of composition, where a given class *contains* another class, we here refer to *derivation*, where a given class *is* or *is-implemented-in-terms-of* another class.

Another term for derivation is *inheritance*: the new class inherits the functionality of an existing class, while the existing class does not appear as a data member in the interface of the new class. When discussing inheritance the existing class is called the *base class*, while the new class is called the *derived class*.

Derivation of classes is often used when the methodology of **C++** program development is fully exploited. In this chapter we will first address the syntactic possibilities offered by **C++** for deriving classes. Following this we will address some of the specific possibilities offered by class derivation (inheritance).

As we have seen in the introductory chapter (see section 2.4), in the object-oriented approach to problem solving classes are identified during the problem analysis. Under this approach objects of the defined classes represent entities that can be observed in the problem at hand. The classes are placed in a hierarchy, with the top-level class containing limited functionality. Each new derivation (and hence descent in the class hierarchy) adds new functionality compared to yet existing classes.

In this chapter we shall use a simple vehicle classification system to build a hierarchy of classes. The first class is `Vehicle`, which implements as its functionality the possibility to set or retrieve the weight of a vehicle. The next level in the object hierarchy are land-, water- and air vehicles.

The initial object hierarchy is illustrated in Figure 13.1.

This chapter mainly focuses on the technicalities of class derivation. The distinction between inheritance used to create derived classes whose objects should be considered objects of the base class and inheritance used to implement derived classes *in-terms-of* their base classes will be postponed until the next chapter (14).

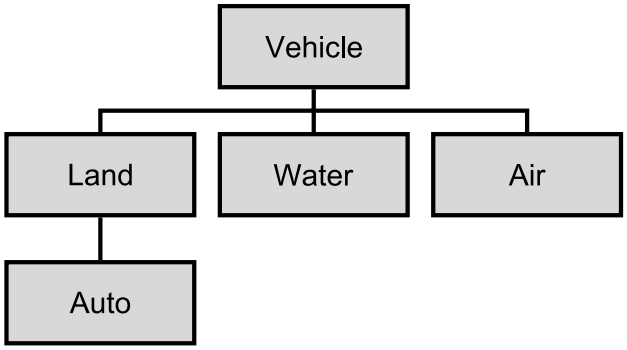

Figure 13.1: Initial object hierarchy of vehicles.

13.1 Related types

The relationship between the proposed classes representing different kinds of vehicles is further investigated here. The figure shows the object hierarchy: an `Auto` is a special case of a `Land` vehicle, which in turn is a special case of a `Vehicle`.

The class `Vehicle` represents the 'greatest common divisor' in the classification system. `Vehicle` is given limited functionality: it can store and retrieve a vehicle's weight:

```
class Vehicle
{
    size_t d_weight;

    public:
        Vehicle();
        Vehicle(size_t weight);

        size_t weight() const;
        void setWeight(size_t weight);
};
```

Using this class, the vehicle's weight can be defined as soon as the corresponding object has been created. At a later stage the weight can be changed or retrieved.

To represent vehicles travelling over land, a new class `Land` can be defined offering `Vehicle`'s functionality and adding its own specific functionality. Assume we are interested in the speed of land vehicles *and* in their weight. The relationship between `Vehicle`s and `Land`s could of course be represented by composition but that would be awkward: composition suggests that a `Land` vehicle *is-implemented-in-terms-of*, i.e., *contains*, a `Vehicle`, while the natural relationship clearly is that the `Land` vehicle *is* a kind of `Vehicle`.

A relationship in terms of composition would also somewhat complicate our `Land` class's design. Consider the following example showing a class `Land` using composition (only the `setWeight` functionality is shown):

```
class Land
{
    Vehicle d_v;        // composed Vehicle
    public:
        void setWeight(size_t weight);
};
```

```
void Land::setWeight(size_t weight)
{
    d_v.setWeight(weight);
}
```

Using composition, the `Land::setWeight` function only passes its argument on to `Vehicle::setWeight`. Thus, as far as weight handling is concerned, `Land::setWeight` introduces no extra functionality, just extra code. Clearly this code duplication is superfluous: a `Land` object *is* a `Vehicle`; to state that a `Land` object *contains* a `Vehicle` is at least somewhat peculiar.

The intended relationship is represented better by inheritance. A rule of thumb for choosing between inheritance and composition distinguishes between *is-a* and *has-a* relationships. A truck *is* a vehicle, so `Truck` should probably derive from `Vehicle`. On the other hand, a truck *has* an engine; if you need to model engines in your system, you should probably express this by composing an `Engine` class inside the `Truck` class.

Following the above rule of thumb, `Land` is *derived* from the base class `Vehicle`:

```
class Land: public Vehicle
{
    size_t d_speed;
    public:
        Land();
        Land(size_t weight, size_t speed);

        void setspeed(size_t speed);
        size_t speed() const;
};
```

To derive a class (e.g., `Land`) from another class (e.g., `Vehicle`) postfix the class name `Land` in its interface by `: public Vehicle`:

```
class Land: public Vehicle
```

The class `Land` now contains all the functionality of its base class `Vehicle` as well as its own features. Here those features are a constructor expecting two arguments and member functions to access the `d_speed` data member. Here is an example showing the possibilities of the derived class `Land`:

```
Land veh(1200, 145);

int main()
{
    cout << "Vehicle weighs " << veh.weight() << ";\n"
            "its speed is " << veh.speed() << '\n';
}
```

This example illustrates two features of derivation.

- First, `weight` is not mentioned as a member in `Land`'s interface. Nevertheless it is used in `veh.weight`. This member function is an implicit part of the class, inherited from its 'parent' vehicle.
- Second, although the derived class `Land` contains the functionality of `Vehicle`, the `Vehicle`'s private members remain private: they can only be accessed by `Vehicle`'s own member functions. This means that `Land`'s member functions *must* use `Vehicle`'s member functions (like `weight` and `setWeight`) to address the `weight` field. Here there's no difference between the access rights granted to `Land` and the access rights granted to other code outside of the class `Vehicle`. The class `Vehicle` *encapsulates* the specific `Vehicle` characteristics, and data hiding is one way to realize encapsulation.

Encapsulation is a core principle of good class design. Encapsulation reduces the dependencies among classes improving the maintainability and testability of classes and allowing us to modify classes without the need to modify depending code. By strictly complying with the principle of data hiding a class's internal data organization may change without requiring depending code to be changed as well. E.g., a class `Lines` originally storing **C**-strings could at some point have its data organization changed. It could abandone its `char **` storage in favor of a `vector<string>` based storage. When `Lines` uses perfect data hiding depending source code may use the new `Lines` class without requiring any modification at all.

As a rule of thumb, derived classes must be fully recompiled (but don't have to be modified) when the *data organization* (i.e., the data members) of their base classes change. Adding new member *functions* to the base class doesn't alter the data organization so no recompilation is needed when new member *functions* are added.

There is one subtle exception to this rule of thumb: if a new member function is added to a base class and that function happens to be the first *virtual* member function of the base class (cf. chapter 14 for a discussion of the virtual member function concept) then that will *also* change the data organization of the base class.

Now that `Land` has been derived from `Vehicle` we're ready for our next class derivation. We'll define a class `Auto` to represent automobiles. Agreeing that an `Auto` object is a `Land` vehicle, and that an `Auto` has a brand name it's easy to design the class `Auto`:

```
class Auto: public Land
{
    std::string d_brandName;

    public:
        Auto();
        Auto(size_t weight, size_t speed, std::string const &name);

        std::string const &brandName() const;
};
```

In the above class definition, `Auto` was derived from `Land`, which in turn is derived from `Vehicle`. This is called *nested derivation*: `Land` is called `Auto`'s *direct base class*, while `Vehicle` is called `Auto`'s *indirect base class*.

13.1.1 Inheritance depth: desirable?

Now that `Auto` has been derived from `Land` and `Land` has been derived from `Vehicle` we might easily be seduced into thinking that these class hierarchies are the way to go when designing classes. But maybe we should temper our enthousiasm.

Repeatedly deriving classes from classes quickly results in big, complex class hierarchies that are hard to understand, hard to use and hard to maintain. Hard to understand and use as users of our derived class now also have to learn all its (indirect) base class features as well. Hard to maintain because all those classes are very closely coupled. While it may be true that when data hiding is meticulously adhered to derived classes do not have to be modified when their base classes alter their data organization, it also quickly becomes practically infeasible to change those base classes once more an more (derived) classes depend on their current organization.

What initially looks like a big gain, inheriting the base class's interface, thus becomes a liability. The base class's interface is hardly ever completely required and in the end a class may benefit from explicitly defining its own member functions rather than obtaining them through inheritance.

Often classes can be defined *in-terms-of* existing classes: some of their features are used, but others need to be shielded off. Consider the `stack` container: it is commonly implemented in-terms-of a `deque`, returning `deque::back`'s value as `stack::top`'s value.

When using inheritance to implement an `is-a` relationship make sure to get the 'direction of use' right: inheritance aiming at implementing an *is-a* relationship should focus on the base class: the base class facilities aren't there to be used by the derived class, but the derived class facilities should redefine (reimplement) the base class facilities using polymorphism (which is the topic of the next chapter), allowing code to use the derived class facilities polymorphically through the base class. We've seen this approach when studying streams: the base class (e.g., `ostream`) is used time and again. The facilities defined by classes derived from `ostream` (like `ofstream` and `ostringstream`) are then used by code only relying on the facilities offered by the `ostream` class, never using the derived classes directly.

When designing classes always aim at the lowest possible coupling. Big class hierarchies usually indicate poor understanding of robust class design. When a class's interface is only partially used and if the derived class is implemented in terms of another class consider using composition rather than inheritance and define the appropriate interface members in terms of the members offered by the composed objects.

13.2 The constructor of a derived class

A derived class inherits functionality from its base class (or base classes, as **C++** supports multiple inheritance, cf. section 13.5). When a derived class object is constructed it is built on top of its base class object. As a consequence the base class must have been constructed before the actual derived class elements can be initialized. This results in some requirements that must be observed when defining derived class constructors.

A constructor exists to initialize the object's data members. A derived class constructor is also responsible for the proper initialization of its base class. Looking at the definition of the class `Land` introduced earlier (section 13.1), its constructor could simply be defined as follows:

```
Land::Land(size_t weight, size_t speed)
{
    setWeight(weight);
    setspeed(speed);
}
```

However, this implementation has several disadvantages.

- When constructing a derived class object a base class constructor will *always* be called before any action is performed on the derived class object itself. By default the base class's default constructor will be called.
- Using the base class constructor only to reassign new values to its data members in the derived class constructor's body usually is inefficient, but sometimes sheer impossible as in situations where base class reference or const data members must be initialized. In those cases a specialized base class constructor must be used instead of the base class default constructor.

A derived class's base class may be initialized using a dedicated base class constructor by calling the base class constructor in the derived class constructor's initializer clause. Calling a base class constructor in a constructor's initializer clause is called a *base class initializer*. The base class initializer must be called before initializing any of the derived class's data members and when using the base class initializer none of the derived class data members may be used. When constructing a derived class object the base class is constructed first and only after that construction has successfully completed the derived class data members are available for initialization. `Land`'s constructor may therefore be improved:

```
Land::Land(size_t weight, size_t speed)
:
    Vehicle(weight),
```

```
    d_speed(speed)
{}
```

Derived class constructors always by default call their base class's default constructor. This is of course not correct for a derived class's copy constructor. Assuming that the class `Land` must be provided with a copy constructor it may use the `Land const &other` to represent the other's base class:

```
Land::Land(Land const &other)   // assume a copy constructor is needed
:
    Vehicle(other),             // copy-construct the base class part.
    d_speed(other.speed)        // copy-construct Land's data members
{}
```

Likewise, a derived class move constructor may have to be defined. A derived class may offer a move constructor for two reasons:

- it supports move construction for its data members
- its base class is move-aware

The design of move constructors moving data members was covered in section 8.6. A move constructor for a derived class whose base class is move-aware must *anonimize* the rvalue reference before passing it to the base class move constructor. The `std::move` function should be used when implementing the move constructor to move the information in base classes or composed objects to their new destination object.

The first example shows the move constructor for the class `Auto`, assuming it has a movable `char *d_brandName` data member and assuming that `Land` is a move-aware class. The second example shows the move constructor for the class `Land`, assuming that it does not itself have movable data members, but that its `Vehicle` base class is move-aware:

```
Auto::Auto(Auto const &&tmp)
:
    Land(std::move(tmp)),           // anonimize 'tmp'
    d_brandName(tmp.d_brandName)    // move the char *'s value
{
    const_cast<Auto &>(tmp).d_brandName = 0;
}

Land(Land const &&tmp)
:
    Vehicle(std::move(tmp)),    // move-aware Vehicle
    d_speed(tmp.d_speed)        // plain copying of plain data
{}
```

13.2.1 Merely using base class constructors (C++0x, ?)

The C++0x standard allows derived classes to be constructed without explicitly defining derived class constructors. In those cases the available base class constructors are called.

This feature is either used or not. It is not possible to omit some of the derived class constructors, using the corresponding base class constructors instead. To use this feature for classes that are derived from multiple base classes (cf. section 13.5) all the base class constructors must have different signatures. Considering the complexities that are involved here it's probably best to avoid using base class constructors for classes using multiple inheritance.

The construction of derived class objects can be delegated to base class constructor(s) using the following syntax:

```
class BaseClass
{
    public:
        // BaseClass constructor(s)
};

class DerivedClass: public BaseClass
{
    public:
        using BaseClass::BaseClass; // No DerivedClass constructors
                                    // are defined
};
```

13.3 The destructor of a derived class

Destructors of classes are automatically called when an object is destroyed. This also holds true for objects of classes derived from other classes. Assume we have the following situation:

```
class Base
{
    public:
        ~Base();
};

class Derived: public Base
{
    public:
        ~Derived();
};

int main()
{
    Derived derived;
}
```

At the end of `main`, the `derived` object ceases to exists. Hence, its destructor (`~Derived`) is called. However, since `derived` is also a `Base` object, the `~Base` destructor is called as well. The base class destructor is never explicitly called from the derived class destructor.

Constructors and destructors are called in a stack-like fashion: when `derived` is constructed, the appropriate base class constructor is called first, then the appropriate derived class constructor is called. When the object `derived` is destroyed, its destructor is called first, automatically followed by the activation of the `Base` class destructor. A derived class destructor is always called before its base class destructor is called.

When the construction of a derived class objects did not successfully complete (i.e., the constructor threw an exception) then its destructor is not called. However, the destructors of properly constructed base classes *will* be called if a derived class constructor throws an exception. This, of course, is it should be: a properly constructed object will also be destroyed, eventually. Example:

```
#include <iostream>
struct Base
{
    ~Base()
    {
        std::cout << "Base destructor\n";
    }
```

```
};
struct Derived: public Base
{
    Derived()
    {
        throw 1;    // at this time Base has been constructed
    }
};
int main()
{
    try
    {
        Derived d;
    }
    catch(...)
    {}
}
/*
    This program displays 'B destructor'
*/
```

13.4 Redefining member functions

Derived classes may redefine base class members. Let's assume that a vehicle classification system must also cover trucks, consisting of two parts: the front part, the tractor, pulls the rear part, the trailer. Both the tractor and the trailer have their own weights, and the `weight` function should return the combined weight.

The definition of a `Truck` starts with a class definition. Our initial `Truck` class is derived from `Auto` but it is then expanded to hold one more `size_t` field representing the additional weight information. Here we choose to represent the weight of the front part of the truck in the `Auto` class and to store the weight of the trailer in an additional field:

```
class Truck: public Auto
{
    size_t d_trailer_weight;

    public:
        Truck();
        Truck(size_t tractor_wt, size_t speed, char const *name,
              size_t trailer_wt);

        void setWeight(size_t tractor_wt, size_t trailer_wt);
        size_t weight() const;
};

Truck::Truck(size_t tractor_wt, size_t speed, char const *name,
             size_t trailer_wt)
:
    Auto(tractor_wt, speed, name)
{
    d_trailer_weight = trailer_wt;
}
```

Note that the class `Truck` now contains two functions already present in the base class `Auto`: `setWeight` and `weight`.

- The redefinition of `setWeight` poses no problems: this function is simply redefined to perform actions which are specific to a `Truck` object.

- The redefinition of `setWeight`, however, will *hide* `Auto::setWeight`. For a `Truck` only the `setWeight` function having two `size_t` arguments can be used.

- The `Vehicle`'s `setWeight` function remains available for a `Truck`, but it must now be called *explicitly*, as `Auto::setWeight` is hidden from view. This latter function is hidden, even though `Auto::setWeight` has only one `size_t` argument. To implement `Truck::setWeight` we could write:

```
void Truck::setWeight(size_t tractor_wt, size_t trailer_wt)
{
    d_trailer_weight = trailer_wt;
    Auto::setWeight(tractor_wt);     // note: Auto:: is required
}
```

- Outside of the class `Auto::setWeight` is accessed using the scope resolution operator. So, if a `Truck truck` needs to set its `Auto` weight, it must use

```
truck.Auto::setWeight(x);
```

- An alternative to using the scope resolution operator is to add a member having the same function prototype as the base class member to the derived class's interface. This derived class member could be implemented inline to call the base class member. E.g., we add the following member to the `class Truck`:

```
// in the interface:
void setWeight(size_t tractor_wt);

// below the interface:
inline void Truck::setWeight(size_t tractor_wt)
{
    Auto::setWeight(tractor_wt);
}
```

  Now the single argument `setWeight` member function can be used by `Truck` objects without using the scope resolution operator. As the function is defined inline, no overhead of an additional function call is involved.

- To prevent hiding the base class members a `using` declaration may be added to the derived class interface. The relevant section of `Truck`'s class interface then becomes:

```
class Truck: public Auto
{
    public:
        using Auto::setWeight;
        void setWeight(size_t tractor_wt, size_t trailer_wt);
};
```

  A using declaration imports (all overloaded versions of) the mentioned member function directly into the derived class's interface. If a base class member has a signature that is identical to a derived class member then compilation will fail (a `using Auto::weight` declaration cannot be added to `Truck`'s interface). Now code may use `truck.setWeight(5000)` as well as `truck.setWeight(5000, 2000)`.

  Using declarations obey access rights. To prevent non-class members from using `setWeight(5000)` without a scope resultion operator but allowing derived class members to do so the `using Auto::setWeight` declaration should be put in the class `Truck`'s private section.

- The function `weight` is also already defined in `Auto`, as it was inherited from `Vehicle`. In this case, the class `Truck` should *redefine* this member function to allow for the extra (trailer) weight in the `Truck`:

```
size_t Truck::weight() const
{
    return
        (                           // sum of:
            Auto::weight() +     //   tractor part plus
            d_trailer_weight        //   the trailer
        );
}
```

Example:

```
int main()
{
    Land veh(1200, 145);
    Truck lorry(3000, 120, "Juggernaut", 2500);

    lorry.Vehicle::setWeight(4000);

    cout << endl << "Truck weighs " <<
                    lorry.Vehicle::weight() << endl <<
        "Truck + trailer weighs " << lorry.weight() << endl <<
        "Speed is " << lorry.speed() << endl <<
        "Name is " << lorry.name() << endl;
}
```

The class `Truck` was derived from `Auto`. However, one might question this class design. Since a truck is conceived of as a combination of an tractor and a trailer it is probably better defined using composition. This changes our point of view from a `Truck` *being* an `Auto` (and some strangely appearing data members) to a `Truck` *consisting of* an `Auto` (the tractor) and a `Vehicle` (the trailer). `Truck`'s interface will be very specific, not requiring users to study `Auto`'s and `Vehicle`'s interfaces and it opens up possibilities for defining 'road trains': tractors towing multiple trailers. Here is an example of such an alternate class setup:

```
class Truck
{
    Auto d_lorry;
    Vehicle d_trailer;      // use vector<Vehicle> for road trains

    public:
        Truck();
        Truck(size_t tractor_wt, size_t speed, char const *name,
              size_t trailer_wt);

        void setWeight(size_t tractor_wt, size_t trailer_wt);
        void setTractorWeight(size_t tractor_wt);
        void setTrailerWeight(size_t trailer_wt);
        size_t weight() const;
        size_t tractorWeight() const;
        size_t trailerWeight() const;
    // consider:
        Auto const &tractor() const;
        Vehicle const &trailer() const;
};
```

13.5 Multiple inheritance

Up to now, a class has always been derived from a single base class. In addition to single inheritance **C++** also supports *multiple inheritance*. In multiple inheritance a class is derived from several base classes and hence inherits functionality from multiple parent classes at the same time.

When using multiple inheritance it should be defensible to consider the newly derived class an instantiation of both base classes. Otherwise, composition is more appropriate. In general, linear derivation (using only one base class) is used much more frequently than multiple derivation. Good class design dictates that a class should have a single, well described responsibility and that principle often conflicts with multiple inheritance where we can state that objects of class `Derived` are *both* `Base1` *and* `Base2` objects.

But then, consider *the* prototype of an object for which multiple inheritance was used to its extreme: the *Swiss army knife*! This object *is* a knife, it *is* a pair of scissors, it *is* a can-opener, it *is* a corkscrew, it *is* ....

The 'Swiss army knife' is an extreme example of multiple inheritance. In **C++** there *are* some good reasons, not violating the 'one class, one responsibility' principle that will be covered in the next chapter. In this section the technical details of constructing classes using multiple inheritance are discussed.

How to construct a 'Swiss army knife' in **C++**? First we need (at least) two base classes. For example, let's assume we are designing a toolkit allowing us to construct an instrument panel of an aircraft's cockpit. We design all kinds of instruments, like an artificial horizon and an altimeter. One of the components that is often seen in aircraft is a *nav-com set*: a combination of a navigational beacon receiver (the 'nav' part) and a radio communication unit (the 'com'-part). To define the nav-com set, we start by designing the `NavSet` class (assume the existence of the classes `Intercom, VHF_Dial` and `Message`):

```
class NavSet
{
    public:
        NavSet(Intercom &intercom, VHF_Dial &dial);

        size_t activeFrequency() const;
        size_t standByFrequency() const;

        void setStandByFrequency(size_t freq);
        size_t toggleActiveStandby();
        void setVolume(size_t level);
        void identEmphasis(bool on_off);
};
```

Next we design the class `ComSet`:

```
class ComSet
{
    public:
        ComSet(Intercom &intercom);

        size_t frequency() const;
        size_t passiveFrequency() const;

        void setPassiveFrequency(size_t freq);
        size_t toggleFrequencies();

        void setAudioLevel(size_t level);
        void powerOn(bool on_off);
        void testState(bool on_off);
```

```
        void transmit(Message &message);
};
```

Using objects of this class we can receive messages, transmitted though the `Intercom`, but we can also *transmit* messages using a `Message` object that's passed to the `ComSet` object using its `transmit` member function.

Now we're ready to construct our `NavCom` set:

```
class NavComSet: public ComSet, public NavSet
{
    public:
        NavComSet(Intercom &intercom, VHF_Dial &dial);
};
```

Done. Now we have defined a `NavComSet` which is *both* a `NavSet` *and* a `ComSet`: the facilities of both base classes are now available in the derived class using multiple inheritance.

Please note the following:

- The keyword `public` is present before both base class names (`NavSet` and `ComSet`). By default inheritance uses *private derivation* and the keyword `public` must be repeated before each of the base class specifications. Base classes are not required to use the same derivation type. One base class could have `public` derivation and another base class could use `private` derivation.
- The multiply derived class `NavComSet` introduces no additional functionality of its own, but merely combines two existing classes into a new aggregate class. Thus, **C++** offers the possibility to simply sweep multiple simple classes into one more complex class.
- Here is the implementation of The `NavComSet` constructor:

  ```
  NavComSet::NavComSet(Intercom &intercom, VHF_Dial &dial)
  :
      ComSet(intercom),
      NavSet(intercom, dial)
  {}
  ```

  The constructor requires no extra code: Its purpose is to activate the constructors of its base classes. The order in which the base class initializers are called is *not* dictated by their calling order in the constructor's code, but by the ordering of the base classes in the class interface.
- The `NavComSet` class definition requires no additional data members or member functions: here (and often) the inherited interfaces provide all the required functionality and data for the multiply derived class to operate properly.

Of course, while defining the base classes, we made life easy on ourselves by strictly using different member function names. So, there is a function `setVolume` in the `NavSet` class and a function `setAudioLevel` in the `ComSet` class. A bit cheating, since we could expect that both units in fact have a composed object `Amplifier`, handling the volume setting. A revised class might offer an `Amplifier &amplifier() const` member function, and leave it to the application to set up its own interface to the amplifier. Alternatively, a revised class could define members for setting the volume of either the `NavSet` or the `ComSet` parts.

In situations where two base classes offer identically named members special provisions need to be made to prevent ambiguity:

- The intended base class can explicitly be specified using the base class name and scope resolution operator:

  ```
  NavComSet navcom(intercom, dial);
  ```

```
navcom.NavSet::setVolume(5);    // sets the NavSet volume level
navcom.ComSet::setVolume(5);    // sets the ComSet volume level
```

- The class interface is provided with member functions that can be called unambiguously. These additional members are usually defined `inline`:

```
class NavComSet: public ComSet, public NavSet
{
    public:
        NavComSet(Intercom &intercom, VHF_Dial &dial);
        void comVolume(size_t volume);
        void navVolume(size_t volume);
};
inline void NavComSet::comVolume(size_t volume)
{
    ComSet::setVolume(volume);
}
inline void NavComSet::navVolume(size_t volume)
{
    NavSet::setVolume(volume);
}
```

- If the `NavComSet` class is obtained from a third party, and cannot be modified, a disambiguating wrapper class may be used:

```
class MyNavComSet: public NavComSet
{
    public:
        MyNavComSet(Intercom &intercom, VHF_Dial &dial);
        void comVolume(size_t volume);
        void navVolume(size_t volume);
};
inline MyNavComSet::MyNavComSet(Intercom &intercom, VHF_Dial &dial)
:
    NavComSet(intercom, dial);
{}
inline void MyNavComSet::comVolume(size_t volume)
{
    ComSet::setVolume(volume);
}
inline void MyNavComSet::navVolume(size_t volume)
{
    NavSet::setVolume(volume);
}
```

13.6 Public, protected and private derivation

With inheritance public derivation is frequently used. When public derivation is used the access rights of the base class's interface remains unaltered in the derived class. But the type of inheritance may also be defined as *private* or *protected*.

Protected derivation is used when the keyword `protected` is put in front of the derived class's base class:

```
class Derived: protected Base
```

When protected derivation is used all the base class's public and protected members become protected members in the derived class. The derived class may access all the base class's public and protected members. Classes in turn derived from the derived class will view the base class's members as protected, and other code (outside of the inheritance tree) will not be able to access the base class's members.

Private derivation is used when the keyword `private` is put in front of the derived class's base class:

```
class Derived: private Base
```

When private derivation is used all the base class's members turn into private members in the derived class. The derived class members may access all base class public and protected members but base class members cannot be used elsewhere.

Public derivation should be used to define an `is-a` relationship between a derived class and a base class: the derived class object *is-a* base class object allowing the derived class object to be used polymorphically as a base class object in code expecting a base class object. Private inheritance is used in situations where a derived class object is defined in-terms-of the base class where composition cannot be used. There's little documented use for protected inheritance, but one could maybe encounter protected inheritance when defining a base class that is itself a derived class and needs to make its base class members available to classes derived from itself.

Combinations of inheritance types do occur. For example, when designing a stream-class it is usually derived from `std::istream` or `std::ostream`. However, before a stream can be constructed, a `std::streambuf` must be available. Taking advantage of the fact that the inheritance order is defined in the class interface, we use multiple inheritance (see section 13.5) to derive the class from both `std::streambuf` and (then) from `std::ostream`. To the class's users it is a `std::ostream` and not a `std::streambuf`. So private derivation is used for the latter, and public derivation for the former class:

```
class Derived: private std::streambuf, public std::ostream
```

13.6.1 Promoting access rights

When private or protected derivation is used, users of derived class objects are denied access to the base class members. Private derivation denies access to all base class members to users of the derived class, protected derivation does the same, but allows classes that are in turn derived from the derived class to access the base class's public and protected members.

In some situations this scheme is too restrictive. Consider a class `RandStream` derived privately from a class `RandBuf` which is itself derived from `std::streambuf` and also publicly from `istream`:

```
class RandBuf: public std::streambuf
{
    // implements a buffer for random numbers
};
class RandStream: private RandBuf, public std::istream
{
    // implements a stream to extract random values from
};
```

Such a class could be used to extract, e.g., random numbers using the standard `istream` interface.

Although the `RandStream` class is constructed with the functionality of `istream` objects in mind, some of the members of the class `std::streambuf` may be considered useful by themselves. E.g., the function `streambuf::in_avail` returns a lower bound to the number of characters that can be read immediately. The standard way to make this function available is to define a *shadow member* calling the base class's member:

```
class RandStream: private RandBuf, public std::istream
{
    // implements a stream to extract random values from
    public:
        std::streamsize in_avail();
};
inline std::streamsize RandStream::in_avail()
{
    return std::streambuf::in_avail();
}
```

This looks like a lot of work for just making available a member from the protected or private base classes. If the intent is to make available the `in_avail` member *access promotion* can be used. Access promotion allows us to specify which members of private (or protected) base classes become available in the protected (or public) interface of the derived class. Here is the above example, now using access promotion:

```
class RandStream: private RandBuf, public std::istream
{
    // implements a stream to extract random values from
    public:
        using std::streambuf::in_avail;
};
```

It should be noted that access promotion makes available all overloaded versions of the declared base class member. So, if `streambuf` would offer not only `in_avail` but also, e.g., `in_avail(size_t *)` *both* members would become part of the public interface.

13.7 Conversions between base classes and derived classes

When public inheritance is used to define classes, an object of a derived class *is* at the same time an object of the base class. This has important consequences for object assignment and for the situation where pointers or references to such objects are used. Both situations will be discussed now.

13.7.1 Conversions with object assignments

Continuing our discussion of the `NavCom` class, introduced in section 13.5, we now define two objects, a base class and a derived class object:

```
ComSet com(intercom);
NavComSet navcom(intercom2, dial2);
```

The object `navcom` is constructed using an `Intercom` and a `VHF_Dial` object. However, a `NavComSet` is at the same time a `ComSet`, allowing the assignment *from* `navcom` (a derived class object) *to* `com` (a base class object):

```
com = navcom;
```

The effect of this assignment will be that the object `com` will now communicate with `intercom2`. As a `ComSet` does not have a `VHF_Dial`, the `navcom`'s `dial` is ignored by the assignment. When assigning a base class object from a derived class object only the base class data members are assigned, other data members are dropped, a phenomenon called *slicing*. In situations like these slicing probably does not have serious consequences, but when passing derived class objects to functions defining base class

parameters or when returning derived class objects from functions returning base class objects slicing also occurs and might have unwelcome side-effects.

The assignment from a base class object to a derived class object is problematic. In a statement like

```
navcom = com;
```

it isn't clear how to reassign the `NavComSet`'s `VHF_Dial` data member as they are missing in the `ComSet` object `com`. Such an assignment is therefore refused by the compiler. Although derived class objects are also base class objects, the reverse does not hold true: a base class object is not also a derived class object.

The following general rule applies: in assignments in which base class objects and derived class objects are involved, assignments in which data are dropped are legal (called *slicing*). Assignments in which data remain unspecified are *not* allowed. Of course, it is possible to overload an assignment operator to allow the assignment of a derived class object from a base class object. To compile the statement

```
navcom = com;
```

the class `NavComSet` must have defined an overloaded assignment operator accepting a `ComSet` object for its argument. In that case it's up to the programmer to decide what the assignment operator will do with the missing data.

13.7.2 Conversions with pointer assignments

We return to our `Vehicle` classes, and define the following objects and pointer variable:

```
Land land(1200, 130);
Auto auto(500, 75, "Daf");
Truck truck(2600, 120, "Mercedes", 6000);
Vehicle *vp;
```

Now we can assign the addresses of the three objects of the derived classes to the `Vehicle` pointer:

```
vp = &land;
vp = &auto;
vp = &truck;
```

Each of these assignments is acceptable. However, an implicit conversion of the derived class to the base class `Vehicle` is used, since `vp` is defined as a pointer to a `Vehicle`. Hence, when using `vp` only the member functions manipulating `weight` can be called as this is the `Vehicle`'s *only* functionality. As far as the compiler can tell this is the object `vp` points to.

The same holds true for references to `Vehicles`. If, e.g., a function is defined having a `Vehicle` reference parameter, the function may be passed an object of a class derived from `Vehicle`. Inside the function, the specific `Vehicle` members remain accessible. This analogy between pointers and references holds true in general. Remember that a reference is nothing but a pointer in disguise: it mimics a plain variable, but actually it is a pointer.

This restricted functionality has an important consequence for the class `Truck`. Following `vp = &truck`, `vp` points to a `Truck` object. So, `vp->weight()` will return 2600 instead of 8600 (the combined weight of the cabin and of the trailer: 2600 + 6000), which would have been returned by `truck.weight()`.

When a function is called using a pointer to an object, then the *type of the pointer* (and not the type of the object) determines which member functions are available and will be executed. In other words, **C++** implicitly converts the type of an object reached through a pointer to the pointer's type.

If the actual type of the object pointed to by a pointer is known, an explicit type cast can be used to access the full set of member functions that are available for the object:

```
Truck truck;
Vehicle *vp;

vp = &truck;        // vp now points to a truck object

Truck *trp;

trp = reinterpret_cast<Truck *>(vp);
cout << "Make: " << trp->name() << endl;
```

Here, the second to last statement specifically casts a `Vehicle *` variable to a `Truck *`. As usual (with type casts), this code is not without risk. It will *only* work if `vp` really points to a `Truck`. Otherwise the program may produce unexpected results.

13.8 Using non-default constructors with new[]

An often heard complaint is that operator `new[]` calls the default constructor of a class to initialize the allocated objects. For example, to allocate an array of 10 strings we can do

```
new string[10];
```

but it is not possible to use another constructor. Assuming that we'd want to initialize the strings with the text `hello world`, we can't write something like:

```
new string("hello world")[10];
```

The initialization of a dynamically allocated object usually consists of a two-step process: first the array is allocated (implicitly calling the default constructor); second the array's elements are initialized, as in the following little example:

```
string *sp = new string[10];
fill(sp, sp + 10, string("hello world"));
```

These approaches all suffer from 'double initializations', comparable to not using member initializers in constructors.

One way to avoid double initialization is to use inheritance. Inheritance can profitably be used to call non-default constructors in combination with operator `new[]`. The approach capitalizes on the following:

- A base class pointer may point to a derived class object;
- A derived class without (non-static) data members has the same size as its base class.

The above also suggests s possible approach:

- Derive a simple, member-less class from the class we're interested in;
- Use the appropriate base class initializer in its default constructor;
- Allocate the required number of derived class objects, and assign `new[]`'s return expression to a pointer to base class objects.

Here is a simple example, producing 10 lines containing the text `hello world`:

```
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>

using namespace std;

struct Xstr: public string
{
    Xstr()
    :
        string("hello world")
    {}
};

int main()
{
    string *sp = new Xstr[10];
    copy(sp, sp + 10, ostream_iterator<string>(cout, "\n"));
}
```

Of course, the above example is fairly unsophisticated, but it's easy to polish the example: the class `Xstr` can be defined in an anonymous namespace, accessible only to a function `getString()` which may be given a `size_t nObjects` parameter, allowing users to specify the number of `hello world`-initialized strings they would like to allocate.

Instead of hard-coding the base class arguments it's also possible to use variables or functions providing the appropriate values for the base class constructor's arguments. In the next example a *local class* `Xstr` is defined inside a function `nStrings(size_t nObjects, char const *fname)`, expecting the number of `string` objects to allocate and the name of a file whose subsequent lines are used to initialize the objects. The local class is invisible outside of the function `nStrings`, so no special namespace safeguards are required.

As discussed in section 7.7, members of local classes cannot access local variables from their surrounding function. However, they can access global and static data defined by the surrounding function.

Using a local class neatly allows us to hide the implementation details within the function `nStrings`, which simply opens the file, allocates the objects, and closes the file again. Since the local class is derived from `string`, it can use any `string` constructor for its base class initializer. In this particular case it calls the `string(char const *)` constructor, providing it with subsequent lines of the just opened stream via its static member function `nextLine()`. This latter function is, as it is a static member function, available to `Xstr` default constructor's member initializers even though no `Xstr` object is available by that time.

```
#include <fstream>
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>

using namespace std;

string *nStrings(size_t size, char const *fname)
{
    static ifstream in;

    struct Xstr: public string
```

```
    {
        Xstr()
        :
            string(nextLine())
        {}
        static char const *nextLine()
        {
            static string line;

            getline(in, line);
            return line.c_str();
        }
    };
    in.open(fname);
    string *sp = new Xstr[10];
    in.close();

    return sp;
}

int main()
{
    string *sp = nStrings(10, "nstrings.cc");
    copy(sp, sp + 10, ostream_iterator<string>(cout, "\n"));
}
```

When this program is run, it displays the first 10 lines of the file `nstrings.cc`.

Note that the above implementation can't safely be used in a multithreaded environment. In that case a *mutex* should be used to protect the three statements just before the function's return statement.

A completely different way to avoid the double initialization (not using inheritance) is to use placement new (cf. section 8.1.4): simply allocate the required amount of memory followed by the proper in-place allocation of the objects, using the appropriate constructors. The following example can also be used in multithreaded environments. The approach uses a pair of static `construct/destroy` members to perform the required initialization.

In the program shown below `construct` expects a `istream` that provides the initialization strings for objects of a class `String` simply containing a `std::string` object. `Construct` first allocates enough memory for the `n` `String` objects plus room for an initial `size_t` value. This initial `size_t` value is then initialized with `n`. Next, in a `for` statement, lines are read from the provided stream and the lines are passed to the constructors, using placement new calls. Finally the address of the first `String` object is returned.

The member `destroy` handles the destruction of the objects. It retrieves the number of objects to destroy from the `size_t` it finds just before the location of the address of the first object to destroy. The objects are then destroyed by explicitly calling their destructors. Finally the raw memory, originally allocated by `construct` is returned.

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class String
{
    union Ptrs
    {
        void *vp;
        String *sp;
```

```
        size_t *np;
    };

    std::string d_str;

    public:
        String(std::string const &txt)
        :
            d_str(txt)
        {}
        ~String()
        {
            cout << "destructor: " << d_str << '\n';
        }
        static String *construct(istream &in, size_t n)
        {
            Ptrs p = {operator new(n * sizeof(String) + sizeof(size_t))};
            *p.np++ = n;

            string line;
            for (size_t idx = 0; idx != n; ++idx)
            {
                getline(in, line);
                new(p.sp + idx) String(line);
            }

            return p.sp;
        }
        static void destroy(String *sp)
        {
            Ptrs p = {sp};
            --p.np;
            for (size_t n = *p.np; n--; )
                sp++->~String();

            operator delete (p.vp);
        }
};

int main()
{
    String *sp = String::construct(cin, 5);

    String::destroy(sp);
}

/*
    After providing 5 lines containing, respectively
        alfa, bravo, charlie, delta, echo
    the program displays:
                destructor: alfa
                destructor: bravo
                destructor: charlie
                destructor: delta
                destructor: echo
*/
```

Chapter 14

Polymorphism

Using inheritance classes may be derived from other classes, called base classes. In the previous chapter we saw that base class pointers may be used to point to derived class objects. We also saw that when a base class pointer points to an object of a derived class it is the the type of the pointer rather than the type of the object it points to what determines which member functions are visible. So when a `Vehicle *vp`, points to an `Auto` object `Auto`'s `speed` or `brandName` members can't be used.

In the previous chapter two fundamental ways classes may be related to each other were discussed: a class may be *implemented-in-terms-of* another class and it can be stated that a derived class *is-a* base class. The former relationship is usually implemented using composition, the latter is usually implemented using a special form of inheritance, called *polymorphism*, the topic of this chapter.

An *is-a* relationship between classes allows us to apply the *Liskov Substitution Principle* (*LSP*) according to which a derived class object may be passed to and used by code expecting a pointer or reference to a base class object. In the annotation() so far the LSP has been applied many times. Every time an `ostringstream, ofstream` or `fstream` was passed to functions expecting an `ostream` we've been applying this principle. In this chapter we'll discover how to design our own classes accordingly.

LSP is implemented using a technique called *polymorphism*: although a base class pointer is used it will perform actions defined in the (derived) class of the object it actually points to. So, a `Vehicle *vp` might behave like an `Auto *` when pointing to an `Auto`[1].

Polymorphism is implemented using a feature called *late binding*. It's called that way because the decision *which* function to call (a base class function or a function of a derived class) cannot be made *compile-time*, but is postponed until the program is actually executed: only then it is determined which member function will actually be called.

In **C++** late binding is *not* the default way functions are called. By default *static binding* (or *early binding*) is used. With static binding the functions that are called are determined by the compiler, merely using the class types of objects, object pointers or object refences.

Late binding is an inherently different (and slightly slower) process as it is decided run-time, rather than compile-time what function will be called. As **C++** supports *both* late- and early-binding **C++** programmers are offered an option as to what kind of binding to use. Choices can be optimized to the situations at hand. Many other languages offering object oriented facilities (e.g., **Java**) only or by default offer late binding. **C++** programmers should be keenly aware of this. Expecting early binding and getting late binding may easily produce nasty bugs.

Let's look at a simple example to start appreciating the differences between late and early binding. The example merely illustrates. Explanations of *why* things are as shown will shortly be provided.

Consider the following little program:

[1] In one of the StarTrek movies, Capt. Kirk was in trouble, as usual. He met an extremely beautiful lady who, however, later on changed into a hideous troll. Kirk was quite surprised, but the lady told him: "Didn't you know I am a polymorph?"

```
#include <iostream>
using namespace std;

class Base
{
    protected:
        void hello()
        {
            cout << "base hello\n";
        }
    public:
        void process()
        {
            hello();
        }
};
class Derived: public Base
{
    protected:
        void hello()
        {
            cout << "derived hello\n";
        }
};
int main()
{
    Derived derived;

    derived.process();
}
```

The important characteristic of the above program is the `Base::process` function, calling `hello`. As `process` is the only member that is defined in the public interface it is the only member that can be called by code not belonging to the two classes. The class `Derived`, derived from `Base` clearly inherits `Base`'s interface and so `process` is also available in `Derived`. So the `Derived` object in `main` is able to call `process`, but not `hello`.

So far, so good. Nothing new, all this was covered in the previous chapter. One may wonder why `Derived` was defined at all. It was presumably defined to create an implementation of `hello` that's appropriate for `Derived` but differing from `Base::hello`'s implementation. `Derived`'s author's reasoning was as follows: `Base`'s implementation of `hello` is not appropriate; a `Derived` class object can remedy that by providing an appropriate implementation. Furthermore our author reasoned:

> "since the type of an object determines the interface that is used, `process` must call `Derived::hello` as `hello` is called via `process` from a `Derived` class object".

Unfortunately our author's reasoning is flawed, due to static binding. When `Base::process` was compiled static binding caused the compiler to fixate the `hello` call to `Base::hello()`.

The author *intended* to create a `Derived` class that `is-a` `Base` class. That only partially succeeded: `Base`'s interface was inherited, but after that `Derived` has relinquished all control over what happens. Once we're in `process` we're only able to see `Base`'s member implementations. Polymorphism offers a way out, allowing us to redefine (in a derived class) members of a base class allowing these redefined members to be used from the base class's interface.

This is the essence of LSP: public inheritance should not be used to reuse the base class members (in derived classes) but to be reused (by the base class, polymorphically using derived class members reimplementing base class members).

Take a second to appreciate the implications of the above little program. The `hello` and `process` mem-

bers aren't too impressive, but the implications of the example are. The `process` member could implement directory travel, `hello` could define the action to perform when encountering a file. `Base::hello` might simply show the name of a file, but `Derived::hello` might delete the file; might only list its name if its younger than a certain age; might list its name if it contains a certain text; etc., etc.. Up to now `Derived` would have to implement `process`'s actions itself; Up to now code expecting a `Base` class reference or pointer could only perform `Base`'s actions. Polymorphism allows us to reimplement members of base classes and to use those reimplemented members in code expecting base class references or pointers. Using polymorphism existing code may be reused by derived classes reimplementing the appropriate members of their base classes. It's about time to uncover how this magic can be realized.

Polymorphism, which is not the default in **C++**, solves the problem and allows the author of the classes to reach its goal. For the curious reader: prefix `void hello()` in the `Base` class with the keyword `virtual` and recompile. Running the modified program produces the intended and expected `derived hello`. Why this happens is explained next.

14.1 Virtual functions

By default the behavior of a member function called via a pointer or reference is determined by the implementation of that function in the pointer's or reference's class. E.g., a `Vehicle *` will activate `Vehicle`'s member functions, even when pointing to an object of a derived class. This is known as as *early* or *static* binding: the function to call is determined compile-time. In **C++** *late* or *dynamic* binding is realized using *virtual member functions*.

A member function becomes a virtual member function when its declaration starts with the keyword `virtual`. It is stressed once again that in **C++**, different from several other object oriented languages, this is *not* the default situation. By default *static* binding is used.

Once a function is declared `virtual` in a base class, it remains virtual in all derived classes; even when the keyword `virtual` is not repeated in derived classes.

In the vehicle classification system (see section 13.1) the two member functions `weight` and `setWeight` might be declared `virtual`. Concentrating on `weight` The relevant sections of the class definitions of the class `Vehicle` and `Truck` are shown below. Also, we show the implementations of the member function `weight`:

```
class Vehicle
{
    public:
        virtual int weight() const;
};
class Truck: // inherited from Vehicle through Auto and Land
{
    // not altered
};
int Vehicle::weight() const
{
    return d_weight;
}

int Truck::weight() const
{
    return Auto::weight() + d_trailer_wt;
}
```

The keyword `virtual` *only* appears in the (`Vehicle`) base class. There is no need (but there is also no penalty) to repeat it in derived classes. Once a class member has been declared `virtual` is will be `virtual` in all derived classes. A member function may be declared `virtual` *anywhere* in a class hierarchy. The compiler will be perfectly happy if `weight` is declared `virtual` in `Auto`, rather than

in Vehicle. The specific characteristics of virtual member functions would then only be available for Auto objects and for objects of classes derived from Auto. For a Vehicle pointer static binding would remain to be used. The effect of late binding is illustrated below:

```
Vehicle v(1200);            // vehicle with weight 1200
Truck t(6000, 115,          // truck with cabin weight 6000, speed 115,
      "Scania", 15000);     // make Scania, trailer weight 15000
Vehicle *vp;                // generic vehicle pointer

int main()
{
    vp = &v;                            // see (1) below
    cout << vp->weight() << endl;

    vp = &t;                            // see (2) below
    cout << vp->weight() << endl;

    cout << vp->speed() << endl;     // see (3) below
}
```

Now that weight is defined virtual, late binding will be used:

- at (1), Vehicle::weight is called.
- at (2) Truck::weight is called.
- at (3) a syntax error is generated. The member speed is no member of Vehicle, and hence not callable via a Vehicle*.

The example illustrates that when a pointer to a class is used *only the members of that class can be called*. These functions may or may not be virtual. A member's virtual characteristic only influences the type of binding (early vs. late), not the set of member functions that is visible to the pointer.

Through virtual members derived classes may redefine the behavior performed by functions called from base class members or from pointers or references to base class objects. This redefinition of base class members by derived classes is called *overriding members*.

14.2 Virtual destructors

When an object ceases to exist the object's destructor is called. Now consider the following code fragment (cf. section 13.1):

```
Vehicle *vp = new Land(1000, 120);

delete vp;          // object destroyed
```

Here delete is applied to a base class pointer. As the base class defines the available interface delete vp calls ~Vehicle and ~Land remains out of sight. Assuming that Land allocates memory a memory leak results. Freeing memory is not the only action destructors can perform. In general they may perform any action that's necessary when an object ceases to exist. But here none of the actions defined by ~Land will be performed. Bad news....

In **C++** this problem is solved by *virtual destructors*. A destructor can be declared virtual. When a base class destructor is declared virtual the destructor of the actual class pointed to by a base class

pointer `bp` will be called when executing `delete bp`. Thus, late binding is realized for destructors even though the destructors of derived classes have unique names. Example:

```
class Vehicle
{
    public:
        virtual ~Vehicle();     // all derived class destructors are
                                // now virtual as well.
};
```

By declaring a virtual destructor, the above `delete` operation (`delete vp`) will correctly call `Land`'s destructor, rather than `Vehicle`'s destructor.

Once a destructor is called it will perform as usual, whether or not it is a virtual destructor. So, `~Land` will first execute its own statements and will then call `~Vehile`. Thus, the above `delete vp` statement will use late binding to call `~Vehicle` and from this point on the object destruction proceeds as usual.

Destructors should always be defined `virtual` in classes designed as a base class from which other classes are going to be derived. Often those destructors themselves have no tasks to perform. In these cases the virtual destructor is given an empty body. For example, the definition of `Vehicle::~Vehicle()` may be as simple as:

```
Vehicle::~Vehicle()
{}
```

Resist the temptation to define destructors (even empty destructors) inline as this will complicate class maintenance. Section 14.9 discusses the reason behind this rule of thumb.

14.3 Pure virtual functions

The base class `Vehicle` is provided with its own concrete implementations of its virtual members (`weight` and `setWeight`). However, virtual member functions not necessarily *have* to be implemented in base classes.

When the implementations of virtual members are omitted from base classes the class imposes requirements upon derived classes. The derived classes are required to provide the 'missing implementations'

This approach, in some languages (like **C#, Delphi** and **Java**) known as an *interface*, defines a *protocol*. Derived classes *must* obey the protocol by implementing the as yet not implemented members. If a class contains at least one member whose implementation is missing no objects of that class can be defined.

Such incompletely defined classes are always base classes. They enforce a protocol by merely declaring names, return values and arguments of some of their members. These classes are call *abstract classes* or *abstract base classes*. Derived classes become non-abtract classes by implementing the as yet not implemented members.

Abstract base classes are the foundation of many *design patterns* (cf. *Gamma et al.* (1995)) , allowing the programmer to create highly *reusable software*. Some of these design patterns are covered by the **C++** Annotations (e.g, the *Template Method* in section 23.4), but for a thorough discussion of design patterns the reader is referred to Gamma *et al.*'s book.

Members that are merely declared in base classes are called *pure virtual functions*. A virtual member becomes a pure virtual member by postfixing `= 0` to its declaration (i.e., by replacing the semicolon ending its declaration by '`= 0;`'). Example:

```
#include <iosfwd>
class Base
```

```
    {
        public:
            virtual ~Base();
            virtual std::ostream &insertInto(std::ostream &out) const = 0;
    };
    inline std::ostream &operator<<(std::ostream &out, Base const &base)
    {
        return base.insertInto(out);
    }
```

All classes derived from `Object` *must* implement the `insertInto` member function, or their objects cannot be constructed. This is neat: all objects derived from `Object` can now always be inserted into `ostream` objects.

Could the virtual destructor of a base class ever be a pure virtual function? The answer to this question is no. First of all, there is no need to enforce the availability of destructors in derived classes as destructors are provided by default (unless a destructor is declared with the `= delete` attribute using the new C++0x standard). Second, if it is a pure virtual member its implementation does not exist, but derived class destructors will eventually call their base class destructors. How could they call base class destructors if their implementations are lacking? More about this in the next section.

Often, but not necessarily, pure virtual member functions are `const` member functions. This allows the construction of constant derived class objects. In other situations this might not be necessary (or realistic), and non-constant member functions might be required. The general rule for `const` member functions also applies to pure virtual functions: if the member function alters the object's data members, it cannot be a `const` member function.

Abstract base classes frequently don't have data members. However, once a base class declares a pure virtual member it *must* be declared identically in derived classes. If the implementation of a pure virtual function in a derived class alters the derived class object's data, then *that* function cannot be declared as a `const` member. Therefore, the author of an abstract base class should carefully consider whether a pure virtual member function should be a `const` member function or not.

14.3.1 Implementing pure virtual functions

Pure virtual member functions may be implemented. To implement a pure virtual member function: pure virtual implementation member function, provide it with its normal `= 0;` specification, but implement it as well. Since the `= 0;` ends in a semicolon, the pure virtual member is always at most a declaration in its class, but an implementation may either be provided outside from its interface (maybe using `inline`).

Pure virtual member functions may be called from derived class objects or from its class or derived class members by specifying the base class and scope resolution operator together with the member to be called. Example:

```
#include <iostream>

class Base
{
    public:
        virtual ~Base();
        virtual void pureimp() = 0;
};
Base::~Base()
{}
void Base::pureimp()
{
    std::cout << "Base::pureimp() called\n";
```

```
}
class Derived: public Base
{
    public:
        virtual void pureimp();
};
inline void Derived::pureimp()
{
    Base::pureimp();
    std::cout << "Derived::pureimp() called\n";
}
int main()
{
    Derived derived;

    derived.pureimp();
    derived.Base::pureimp();

    Derived *dp = &derived;

    dp->pureimp();
    dp->Base::pureimp();
}
// Output:
//      Base::pureimp() called
//      Derived::pureimp() called
//      Base::pureimp() called
//      Base::pureimp() called
//      Derived::pureimp() called
//      Base::pureimp() called
```

Implementing a pure virtual member has limited use. One could argue that the pure virtual member function's implementation may be used to perform tasks that can already be performed at the base class level. However, there is no guarantee that the base class virtual member function will actually be called. Therefore a base class specific tasks could as well be offered by a separate member, without blurring the distinction between a member doing some work and a pure virtual member enforcing a protocol.

14.4 Virtual functions and multiple inheritance

In chapter **??** we encountered the class `fstream`, one class offering features of `ifstream` and `ofstream`. In chapter 13 we learned that a class may be derived from multiple base classes. Such a derived class inherits the properties of all its base classes. Polymorphism can also be used in combination with multiple inheritance.

Consider what would happen if more than one 'path' leads from the derived class up to its (base) classes. This is illustrated in the next (fictitious) example where a class `Derived` is doubly derived from `Base`:

```
    class Base
    {
        int d_field;
        public:
            void setfield(int val);
            int field() const;
    };
    inline void Base::setfield(int val)
    {
```

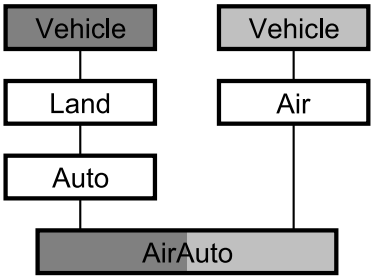

Figure 14.1: Duplication of a base class in multiple derivation.

```
    d_field = val;
}
inline int Base::field() const
{
    return d_field;
}

class Derived: public Base, public Base
{
};
```

Due to the double derivation, `Base`'s functionality now occurs twice in `Derived`. This results in ambiguity: when the function `setfield()` is called for a `Derived` class object, which function will that be as there are two of them? The scope resolution operator won't come to the rescue and so the **C++** compiler will not be able to compile the above example and will (correctly) identify an error.

The above code clearly duplicates its base class in the derivation, which can of course easily be avoided by not doubly deriving from `Base` (or by using composition (!)). But duplication of a base class can also occur through nested inheritance, where an object is derived from, e.g., an `Auto` and from an `Air` (cf. section 13.1). Such a class would be needed to represent, e.g., a flying car[2]. An `AirAuto` would ultimately contain two `Vehicles`, and hence two `weight` fields, two `setWeight()` functions and two `weight()` functions. Is this what we want?

14.4.1 Ambiguity in multiple inheritance

Let's investigate closer why an `AirAuto` introduces ambiguity, when derived from `Auto` and `Air`.

- An `AirAuto` is an `Auto`, hence a `Land`, and hence a `Vehicle`.
- However, an `AirAuto` is also an `Air`, and hence a `Vehicle`.

The duplication of `Vehicle` data is further illustrated in Figure 14.1. The internal organization of an `AirAuto` is shown in Figure 14.2 The **C++** compiler detects the ambiguity in an `AirAuto` object, and will therefore not compile statements like:

```
AirAuto jBond;
cout << jBond.weight() << '\n';
```

Which member function `weight` to call cannot be determined by the compiler but the programmer has two possibilities to resolve the ambiguity for the compiler:

[2] such as the one in James Bond vs. the Man with the Golden Gun...

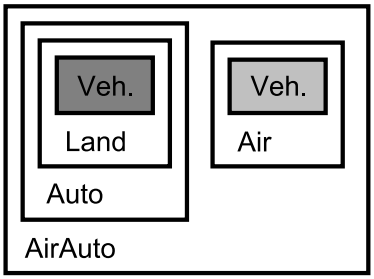

Figure 14.2: Internal organization of an `AirAuto` object.

- First, the function call where the ambiguity originates can be modified. The ambiguity is resolved using the scope resolution operator:

```
// let's hope that the weight is kept in the Auto
// part of the object..
cout << jBond.Auto::weight() << '\n';
```

  The scope resolution operator and the class name are put right before the name of the member function.

- Second, a dedicated function `weight` could be created for the class `AirAuto`:

```
int AirAuto::weight() const
{
    return Auto::weight();
}
```

The second possibility is preferred as it does not require the compiler to flag an error; nor does it require the programmer using the class `AirAuto` to take special precautions.

However, there exists a more elegant solution, discussed in the next section.

14.4.2 Virtual base classes

As illustrated in Figure 14.2, an `AirAuto` represents *two* `Vehicles`. This not only results in an ambiguity about which function to use to access the `weight` data, but it also defines two `weight` fields in an `AirAuto`. This is slightly redundant, since we can assume that an `AirAuto` has but one weight.

It is, however, possible to define an `AirAuto` as a class consisting of but one `Vehicle` and yet using multiple derivation. This is realized by defining the base classes that are multiply mentioned in a derived class's inheritance tree as a *virtual base class*.

For the class `AirAuto` this implies a small change when deriving an `AirAuto` from `Land` and `Air` classes:

```
class Land: virtual public Vehicle
{
    // etc
};
class Auto: public Land
{
    // etc
};
class Air: virtual public Vehicle
```

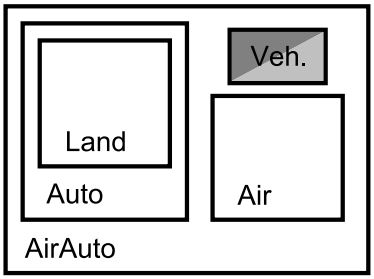

Figure 14.3: Internal organization of an `AirAuto` object when the base classes are virtual.

```
{
    // etc
};
class AirAuto: public Auto, public Air
{
};
```

Virtual derivation ensures that a `Vehicle` is only added once to a derived class. This means that the route along which a `Vehicle` is added to an `AirAuto` is no longer depending on its direct base classes; we can only state that an `AirAuto` is a `Vehicle`. The internal organization of an `AirAuto` after virtual derivation is shown in Figure 14.3.

When a class `Third` inherits from a base class `Second` which in turn inherits from a base class `First` then the `First` class constructor called by the `Second` class constructor is also used when this `Second` constructor is used when constructing a `Third` object. Example:

```
class First
{
    public:
        First(int x);
};
class Second: public First
{
    public:
        Second(int x)
        :
            First(x)
        {}
};
class Third: public Second
{
    public:
        Third(int x)
        :
            Second(x)           // calls First(x)
        {}
};
```

The above no longer holds true when `Second` uses virtual derivation. When `Second` uses virtual derivation its base class constructor is *ignored* when `Second`'s constructor is called from `Third`. Instead `Second` will by default call `First`'s default constructor. This is illustrated by the next example:

When constructing `Third` `First`'s default constructor is used by default. `Third`'s constructor, however, may overrule this default behavior by explicitly specifying the constructor to use. Since the `First` object must be available before `Second` can be constructed it must be specified first. To call `First(int)`

when constructing `Third(int)` the latter constructor can be defined as follows:

```
class Third: public Second
{
    public:
        Third(int x)
        :
            First(x),           // now First(int) is called.
            Second(x)
        {}
};
```

This behavior may seem puzzling when simple linear inheritance is used but it makes sense when multiple inheritance is used with base classes using virtual inheritance. Consider `AirAuto`: when `Air` and `Auto` both virtually inherit from `Vehile` will `Air` and (Auto) both initialize the common `Vehicle` object? If so, which one will be called first? What if `Air` and `Auto` use different `Vehicle` constructors? All these questions can be avoided by passing the responsibility for the initialization of a common base class to the class eventually using the common base class object. In the above example `Third`. Hence `Third` is provided an opportunity to specify the constructor to use when initializing `First`.

Multiple inheritance may also used to inherit from classes that do not all use virtual inheritance. Assume we have two classes, `Derived1` and `Derived2`, both (possibly virtually) derived from `Base`.

We will address the question which constructors will be called when calling a constructor of the class `Final: public Derived1, public Derived2`.

To distinguish the involved constructors `Base1` indicates the `Base` class constructor called as base class initializer for `Derived1` (and analogously: `Base2` called from `Derived2`). A plain `Base` indicates `Base`'s default constructor.

`Derived1` and `Derived2` indicate the base class initializers used when constructing a `Final` object.

Now we're ready to distinguish the various cases when constructing an object of the class `Final: public Derived1, public Derived2`:

- classes:

  ```
  Derived1: public Base
  Derived2: public Base
  ```

  This is normal, non virtual multiple derivation. The following constructors are called in the order shown:

  ```
  Base1,
  Derived1,
  Base2,
  Derived2
  ```

- classes:

  ```
  Derived1: public Base
  Derived2: virtual public Base
  ```

  Only `Derived2` uses virtual derivation. `Derived2`'s base class constructor is ignored. Instead, `Base` will be called and it will be called prior to any other constructor:

  ```
  Base,
  Base1,
  Derived1,
  Derived2
  ```

  As only one class uses virtual derivation, there will still be *two* `Base` class objects in the eventual `Final` class.

- classes:

```
Derived1: virtual public Base
Derived2: public Base
```

  Only `Derived1` uses virtual derivation. `Derived1`'s base class constructor is ignored. Instead, `Base` will be called and it will be called prior to any other constructor. Different from the first (non-virtual) case `Base` is now called, rather than `Base1`:

```
Base,
Derived1,
Base2,
Derived2
```

- classes:

```
Derived1: virtual public Base
Derived2: virtual public Base
```

  Both base classes use virtual derivation and so only *one* `Base` class object will be present in the `Final` class object. The following constructors are called in the order shown:

```
Base,
Derived1,
Derived2
```

Virtual derivation is, in contrast to virtual functions, a pure compile-time issue. Virtual inheritance merely defines how the compiler defines a class's data organization and construction process.

14.4.3 When virtual derivation is not appropriate

Virtual inheritance can be used to merge multiply occurring base classes. However, situations may be encountered where multiple occurrences of base classes is appropriate. Consider the definition of a `Truck` (cf. section 13.4):

```
class Truck: public Auto
{
    int d_trailer_weight;

    public:
        Truck();
        Truck(int engine_wt, int sp, char const *nm,
               int trailer_wt);

        void setWeight(int engine_wt, int trailer_wt);
        int weight() const;
};
Truck::Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt)
:
    Auto(engine_wt, sp, nm)
{
    d_trailer_weight = trailer_wt;
}
int Truck::weight() const
{
    return                  // sum of:
        Auto::weight() +    //   engine part plus
        trailer_wt;         //   the trailer
}
```

This definition shows how a `Truck` object is constructed to contain two weight fields: one via its derivation from `Auto` and one via its own `int d_trailer_weight` data member. Such a definition is of course valid, but it could also be rewritten. We could derive a `Truck` from an `Auto` *and* from a `Vehicle`, thereby explicitly requesting the double presence of a `Vehicle`; one for the weight of the engine and cabin, and one for the weight of the trailer. A slight complication is that a class organization like

```
class Truck: public Auto, public Vehicle
```

is not accepted by the **C++** compiler. As a `Vehicle` is already part of an `Auto`, it is therefore not needed once again. This organzation may, however be forced using a small trick. By creating an additional class inheriting from `Vehicle` and deriving `Truck` from that additional class rather than directly from `Vehicle` the problem is solved. Simply derive a class `TrailerVeh` from `Vehicle`, and then `Truck` from `Auto` and `TrailerVeh`:

```
class TrailerVeh: public Vehicle
{
    public:
        TrailerVeh(int wt)
        :
            Vehicle(wt)
        {}
};
class Truck: public Auto, public TrailerVeh
{
    public:
        Truck();
        Truck(int engine_wt, int sp, char const *nm, int trailer_wt);
        void setWeight(int engine_wt, int trailer_wt);
        int weight() const;
};
inline Truck::Truck(int engine_wt, int sp, char const *nm,
                    int trailer_wt)
:
    Auto(engine_wt, sp, nm),
    TrailerVeh(trailer_wt)
{}
inline int Truck::weight() const
{
    return                      // sum of:
        Auto::weight() +        //   engine part plus
        TrailerVeh::weight();   //   the trailer
}
```

14.5 Run-time type identification

C++ offers two ways to (run-time) retrieve the type of objects and expressions. The possibilities of **C++**'s run-time type identification are limited compared to languages like **Java**. Usually static type checking and static type identification is used in **C++**. Static type checking is possibly safer and certainly more efficient than run-time type identification and should therefore be preferred over run-time type identification. But situations exist where run-time type identification is appropriate. **C++** offers run-time type identification through the *dynamic cast* and `typeid` operators.

- A `dynamic_cast` is used to convert a base class pointer or reference to a derived class pointer or reference. This is also known as *down-casting*.
- The `typeid` operator returns the actual type of an expression.

These operators can be used with objects of classes having at least one virtual member function.

14.5.1 The dynamic_cast operator

The `dynamic_cast<>` operator is used to convert a base class pointer or reference to, respectively, a derived class pointer or reference. This is also called *down-casting* the the direction of the cast is *down* the inheritance tree.

A dynamic cast is performed run-time. A prerequisite for using a dynamic cast is the existence of at least one virtual member function in the base class.

In the following example a pointer to the class `Derived` is obtained from the `Base` class pointer `bp`:

```
class Base
{
    public:
        virtual ~Base();
};
class Derived: public Base
{
    public:
        char const *toString();
};
inline char const *Derived::toString()
{
    return "Derived object";
}
int main()
{
    Base *bp;
    Derived *dp,
    Derived d;

    bp = &d;

    dp = dynamic_cast<Derived *>(bp);

    if (dp)
        cout << dp->toString() << '\n';
    else
        cout << "dynamic cast conversion failed\n";
}
```

In the condition of the above `if` statement the success of the dynamic cast is verified. This verification must be performed *run-time*, as the actual class of the objects to which the pointer points is only known then. If a base class pointer is provided, the dynamic cast operator returns 0 on failure and a pointer to the requested derived class on success.

Assume a `vector<Base *>` is used. Such a vector's pointers may point to objects of various classes, alll derived from `Base`. A dynamic cast returns a pointer to the specified class if the base class pointer indeed points to an object of the specified class and returns 0 otherwise. So we could determine the actual class of an object a pointer points to by performing a series of checks to find the actual derived class to which a base class pointer points. Example:

```
class Base
{
    public:
        virtual ~Base();
```

```
};
class Derived1: public Base;
class Derived2: public Base;

int main()
{
    vector<Base *> vb(initializeBase());

    Base *bp = vb.front();

    if (dynamic_cast<Derived1 *>(bp))
        cout << "bp points to a Derived1 class object\n";
    else if (dynamic_cast<Derived2 *>(bp))
        cout << "bp points to a Derived2 class object\n";
}
```

Alternatively, a reference to a base class object may be available. In this case the `dynamic_cast` operator will throw an exception if the down casting fails. Example:

```
#include <iostream>

class Base
{
    public:
        virtual ~Base();
        virtual char const *toString();
};
inline char const *Base::toString()
{
    return "Base::toString() called";
}
class Derived1: public Base
{};
class Derived2: public Base
{};

Base::~Base()
{}
void process(Base &b)
{
    try
    {
        std::cout << dynamic_cast<Derived1 &>(b).toString() << '\n';
    }
    catch (std::bad_cast)
    {}
    try
    {
        std::cout << dynamic_cast<Derived2 &>(b).toString() << '\n';
    }
    catch (std::bad_cast)
    {
        std::cout << "Bad cast to Derived2\n";
    }
}
int main()
{
    Derived1 d;
    process(d);
```

```
}
/*
    Generated output:

    Base::toString() called
    Bad cast to Derived2
*/
```

In this example the value `std::bad_cast` is used. A `std::bad_cast` exception is thrown if the dynamic cast of a reference to a derived class object fails.

Note the form of the `catch` clause: `bad_cast` is the name of a type. Section 17.4.1 describes how such a type can be defined.

The dynamic cast operator is a useful tool when an existing base class cannot or should not be modified (e.g., when the sources are not available), and a derived class may be modified instead. Code receiving a base class pointer or reference may then perform a dynamic cast to the derived class to access the derived class's functionality.

One may wonder what the difference is between a `dynamic_cast` and a `reinterpret_cast`. One of the differences is of course that the `dynamic_cast` can operate on references while the `reinterpret_cast` can only operate on pointers. But is there a difference when both arguments are pointers?

When the `reinterpret_cast` is used, we tell the compiler that it literally should re-interpret a block of memory as something else. A reinterpret cast could be used to access the individual bytes of an `int`. An `int` consists of `sizeof(int)` bytes, and these bytes can be accessed by reinterpreting the location of the `int` value as a `char *`. When using a `reinterpret_cast` the compiler can no longer offer safeguards against stupidity. The compiler will happily `reinterpret_cast` an `int *` to a `double *`, but the resulting dereference will produce at the very least a questionable value.

The `dynamic_cast` also reinterprets a block of memory, but here a run-time safeguard is provided. The dynamic cast fails when the requested type doesn't match the actual type of the object we're pointing at. The `dynamic_cast`'s purpose is also much more restricted than the `reinterpret_cast`'s purpose, as it can only be used for downcasting to derived classes having virtual members.

In the end a dynamic cast is a cast, and casts should be avoided. When the need for dynamic casting arises ask yourself whether the base class has appropriately been designed. In situations where code expects a base class reference or pointer the base class interface should be all that is required and using a dynamic cast should not be necessary. Maybe the base class's virtual interface can be modified so as to prevent the use of dynamic casts. Start frowning when encountering code using dynamic casts. When using dynamic casts in your own code always properly document why the dynamic cast was appropriately used and could not have been avoided.

14.5.2 The 'typeid' operator

As with the `dynamic_cast` operator, `typeid` is usually applied to references to base class objects that refer to derived class objects. `Typeid` should only be used with base classes offering virtual members. Before using `typeid` the `<typeinfo>` header file must have been included.

The `typeid` operator returns an object of type `type_info`. Different compilers may offer different implementations of the class `type_info`, but at the very least `typeid` must offer the following interface:

```
class type_info
{
    public:
        virtual ~type_info();
        int operator==(type_info const &other) const;
        int operator!=(type_info const &other) const;
        bool before(type_info const &rhs) const
```

```
        char const *name() const;
    private:
        type_info(type_info const &other);
        type_info &operator=(type_info const &other);
};
```

Note that this class has a private copy constructor and a private overloaded assignment operator. This prevents code from constructing `type_info` objects and prevents code from assigning `type_info` objects to each other. Instead, `type_info` objects are constructed and returned by the `typeid` operator.

If the `typeid` operator is passed a base class reference it is able to return the actual name of the type the reference refers to. Example:

```
class Base;
class Derived: public Base;

Derived d;
Base    &br = d;

cout << typeid(br).name() << '\n';
```

In this example the `typeid` operator is given a base class reference. It prints the text "`Derived`", being the class name of the class `br` actually refers to. If `Base` does not contain virtual functions, the text "`Base`" is printed.

The `typeid` operator can be used to determine the name of the actual type of expressions, not just of class type objects. For example:

```
cout << typeid(12).name() << '\n';     // prints:  int
cout << typeid(12.23).name() << '\n';  // prints:  double
```

Note, however, that the above example is suggestive at most. It *may* print `int` and `double`, but this is not necessarily the case. If portability is required, make sure no tests against these static, built-in text-strings are required. Check out what your compiler produces in case of doubt.

In situations where the `typeid` operator is applied to determine the type of a derived class, a base class *reference* should be used as the argument of the `typeid` operator. Consider the following example:

```
class Base;     // contains at least one virtual function
class Derived: public Base;

Base *bp = new Derived;     // base class pointer to derived object

if (typeid(bp) == typeid(Derived *))    // 1: false
    ...
if (typeid(bp) == typeid(Base *))       // 2: true
    ...
if (typeid(bp) == typeid(Derived))      // 3: false
    ...
if (typeid(bp) == typeid(Base))         // 4: false
    ...
if (typeid(*bp) == typeid(Derived))     // 5: true
    ...
if (typeid(*bp) == typeid(Base))        // 6: false
    ...

Base &br = *bp;
```

```
    if (typeid(br) == typeid(Derived))      // 7: true
        ...
    if (typeid(br) == typeid(Base))         // 8: false
        ...
```

Here, `(1)` returns `false` as a `Base *` is not a `Derived *`. `(2)` returns `true`, as the two pointer types are the same, `(3)` and `(4)` return `false` as pointers to objects are not the objects themselves.

On the other hand, if `*bp` is used in the above expressions, then `(1)` and `(2)` return `false` as an object (or reference to an object) is not a pointer to an object, whereas `(5)` now returns `true`: `*bp` actually refers to a `Derived` class object, and `typeid(*bp)` will return `typeid(Derived)`. A similar result is obtained if a base class reference is used: `7` returning `true` and `8` returning `false`.

The `type_info::before(type_info const &rhs)` member is used to determine the *collating order* of classes. This is useful when comparing two *types* for equality. The function returns a nonzero value if `*this` precedes `rhs` in the hierarchy or collating order of the used types. When a derived class is compared to its base class the comparison returns 0, otherwise a non-zero value. E.g.:

```
    cout << typeid(ifstream).before(typeid(istream)) << '\n' << // not 0
            typeid(istream).before(typeid(ifstream)) << '\n';   // 0
```

With built-in types the implementor may implement that non-0 is returned when a 'wider' type is compared to a 'smaller' type and 0 otherwise:

```
    cout << typeid(double).before(typeid(int)) << '\n' <<   // not 0
            typeid(int).before(typeid(double)) << '\n';     // 0
```

When two equal types are compared, 0 is returned:

```
    cout << typeid(ifstream).before(typeid(ifstream)) << '\n';   // 0
```

When a `0`-pointer is passed to the `operator typeid` a `bad_typeid` exception is thrown.

14.6 Inheritance: when to use to achieve what?

Inheritance should not be applied automatically and thoughtlessly. Often composition can be used instead, improving on a class's design by reducing coupling. When inhertitance is used *public* inheritance should not automatically be used but the type of inheritance that is selected should match the programmer's intent.

We've seen that polymorphic classes on the one hand offer interface members defining the functionality that can be requested of base classes and on the other hand offer virtual members that can be overridden. One of the signs of good class design is that member functions are designed according to the principle of 'one function, one task'. In the current context: a class member should either be a member of the class's public or protected interface or it should be available .as a virtual member for reimplementation by derived classes. Often this boils down to virtual members that are defined in the base class's *private* section. Those functions shouldn't be called by code using the base class, but they exist to be overridden by derived classes using polymorphism to redefine the base class's behavior.

The underlying principle was mentioned before in the introductional paragraph of this chapter: according to the *Liskov Substitution Principle* (*LSP*) an *is-a* relationship between classes (indicating that a derived class object *is a* base class object) implies that a derived class object may be used in code expecting a base class object.

In this case inheritance is used *not* to let the derived class use the facilities already implemented by the base class but to reuse the base class polymorphically by reimplementing the base class's virtual members in the derived class.

In this section we'll discuss the reasons for using inheritance. Why should inheritance (not) be used? If it is used what do we try to accomplish by it?

Inheritance often competes with composition. Consider the following two alternative class designs:

```
class Derived: // derived from Base
{ ... };

class Composed
{
    Base d_base;
    ...
};
```

Why and when prefer `Derived` over `Composed` and vice versa? What kind of inheritance should be used when designing the class `Derived`?

- Since `Composed` and `Derived` are offered as alternatives we are looking at the design of a class (`Derived` or `Composed`) that *is-implemented-in-terms-of* another class.
- Since `Composed` does itself not make `Base`'s interface available, `Derived` shouldn't do so either. The underlying principle is that *private inheritance* should be used when deriving from `Base` when `Derived` is-implemented-in-terms-of.
- Should we use inheritance or composition? Here are some arguments:
  - In general terms composition results in looser coupling and should therefore be preferred over inheritance.
  - Composition allows us to define classes having multiple members of the same type (think about a class having multiple `std::string` members) which can not be realized using inheritance.
  - Composition allows us to separate the class's interface from its implementation. This allows us to modify the class's data organization without the need to recompile code using our class. This is also known as the *bridge design pattern* or the *compiler firewal* or *pimpl* (pointer to the implementation) idiom.
  - If `Base` offers members in its *protected* interface that must be used when implementing `Derived` inheritance must also be used. Again: since we're implementing-in-terms-of the inheritance type should be `private`.
  - Protected inheritance may be considered when the derived class (`D`) itself is intended as a base class that should only make the members of its own base class (`B`) available to classes that are derived from it (i.e., `D`).

Private inheritance should also be used when a derived class is-a certain type of base class, but in order to initialize that base class an object of another class type must be available. Example: a new `istream` class-type (say: a stream `IRandStream` from which random numbers can be extracted) will be derived from `std::istream`. Although an `istream` can be constructed empty, receiving its `streambuf` later using its `rdbuf` member it is clearly preferable to initialize the `istream` base class right away.

Assuming that a `Randbuffer: public std::streambuf` has been created for generating random numbers then `IRandStream` can be derived from `Randbuffer` and `std::ostream`. That way the `ostream` base class can be initialized using the `Randbuffer` base class.

As a `RandStream` is definitely not a `Randbuffer` *public* inheritance is *not* appropriate. In this case `IRandStream` is-implemented-in-terms-of a `Randbuffer` and so *private* inheritance should be used.

`IRandStream`'s class interface should therefore start like this:

```
class IRandStream: private Randbuffer, public std::istream
```

```
    {
        public:
            IRandStream(int lowest, int highest)    // defines the range
            :
                Randbuffer(lowest, highest),
                std::istream(this)                  // passes &Randbuffer
            {}
        ...
    };
```

Public inheritance should be reserved for classes for which the LSP holds true. In those cases the derived classes can always be used instead of the base class from which they derive by code merely using base class references, pointers or members (I.e., conceptually the derived class *is-a* base class). This most often applies to classes derived from base classes offering virtual members. To separate the user interface from the redefinable interface the base class's public interface should *not* contain virtual members (except for the virtual destructor) and the virtual members should all be in the base class's private section. Such virtual members can still be overridden by derived classes (this should not come as a surprise, considering how polymorphism is implemented) and this design offers the base class full control over the context in which the redefined members are used. Often the public interface merely calls a virtual member, but those members can always be redefined to perform additional duties.

The prototypical form of a base class therefore looks like this:

```
    class Base
    {
        public:
            virtual ~Base()
            void process();             // calls a virtual member
        private:
            virtual void processImp();  // overridden by derived classes
    };
```

Alternatively a base class may offer a non-virtual destructor, which should then be protected. It shouldn't be public to prevent deleting objects through their base class pointers (in which case virtual destructors should be used). It should be protected to allow derived class destructors to call their base class destructors. Such base classes should, for the same reasons, have non-public constructors and overloaded assignment operators.

14.7 The 'streambuf' class

The class `std::streambuf` receives the character sequences processed by streams and defines the interface between the stream objects and the devices (like a file on disk). A `streambuf` object is usually not directly constructed, but usually it is used as base class of some derived class implementing the communication with some concrete device.

The primary reason for existence of the class `streambuf` is to decouple the `stream` classes from the devices they operate upon. The rationale here is to add an extra layer between the classes allowing us to communicate with devices and the devices themselves. This implements a *chain of command* which is seen regularly in software design.

The *chain of command* is considered a generic pattern when designing reusable software, encountered also in, e.g., the TCP/IP stack.

A `streambuf` can be considered yet another example of the chain of command pattern. Here the program talks to `stream` objects, which in turn forward their requests to `streambuf` objects, which in turn communicate with the devices. Thus, as we will see shortly, we are able to do in user-software what had to be done via (expensive) system calls before.

The class `streambuf` has no public constructor, but does make available several public member functions. In addition to these public member functions, several member functions are only available to classes derived from `streambuf`. In section 14.7.2 a predefined specialization of the class `streambuf` is introduced. All public members of `streambuf` discussed here are *also* available in `filebuf`.

The next section shows the `streambuf` members that may be overridden when deriving classes from `streambuf`. Chapter 23 offers concrete examples of classes derived from `streambuf`.

The class `streambuf` is used by streams performing input operations and by streams performing output operations and their member functions can be ordered likewise. The type `std::streamsize` used below may, for all practical purposes, be considered equal to the type `size_t`.

Public members for input operations

- `std::streamsize in_avail()`:

  This member function returns a lower bound on the number of characters that can be read immediately.

- `int sbumpc()`:

  The next available character or `EOF` is returned. The returned character is removed from the `streambuf` object. If no input is available, `sbumpc` will call the (protected) member `uflow` (see section 14.7.1 below) to make new characters available. `EOF` is returned if no more characters are available.

- `int sgetc()`:

  The next available character or `EOF` is returned. The character is *not* removed from the `streambuf` object.

- `int sgetn(char *buffer, std::streamsize n)`:

  At most `n` characters are retrieved from the input buffer, and stored in `buffer`. The actual number of characters read is returned. The (protected) member `xsgetn` (see section 14.7.1 below) is called to obtain the requested number of characters.

- `int snextc()`:

  The current character is removed from the input buffer and returned as the next available character or `EOF` is returned. The character is *not* removed from the `streambuf` object.

- `int sputback(char c)`:

  Inserts `c` into the `streambuf`'s buffer to be returned as the next character to read from the `streambuf` object. Caution should be exercised when using this function: often there is a maximum of just one character that can be put back.

- `int sungetc()`:

  Returns the last character read to the input buffer, to be read again at the next input operation. Caution should be exercised when using this function: often there is a maximum of just one character that can be put back.

Public members for output operations

- `int pubsync()`:

  Synchronizes (i.e., flush) the buffer by writing any information currently available in the `streambuf`'s buffer to the device. Normally only used by classes derived from `streambuf`.

- `int sputc(char c)`:

  Character `c` is inserted into the `streambuf` object. If, after writing the character, the buffer is full, the function calls the (protected) member function `overflow` to flush the buffer to the device (see section 14.7.1 below).

- `int sputn(char const *buffer, std::streamsize n)`:

  At most `n` characters from `buffer` are inserted into the `streambuf` object. The actual number of characters inserted is returned. This member function calls the (protected) member `xsputn` (see section 14.7.1 below) to insert the requested number of characters.

Public members for miscellaneous operations

The next three members are normally only used by classes derived from `streambuf`.

- `ios::pos_type pubseekoff(ios::off_type offset, ios::seekdir way, ios::openmode mode = ios::in |ios::out)`:

  Sets the offset of the next character to be read or written to `offset`, relative to the standard `ios::seekdir` values indicating the direction of the seeking operation.

- `ios::pos_type pubseekpos(ios::pos_type offset, ios::openmode mode = ios::in |ios::out)`:

  Sets the absolute position of the next character to be read or written to `pos`.

- `streambuf *pubsetbuf(char* buffer, std::streamsize n)`:

  The `streambuf` object will use the `buffer` accomodating at least `n` characters.

14.7.1 Protected 'streambuf' members

The *protected* members of the class `streambuf` are important for understanding and using `streambuf` objects. Athough there are both protected data members and protected member functions defined in the `class streambuf` the protected *data* members are not mentioned here as using them would violates the principle of *data hiding*. As `streambuf`'s set of member functions is quire extensive directly using its data members is probably hardly ever necessary. This section not even lists all protected member functions but lists only those that are useful for constructing specializations.

`Streambuf` objects control a buffer, used for input and/or output, for which begin-, actual- and end-pointers have been defined, as depicted in figure 14.4.

`Streambuf` offers one protected constructor:

- `streambuf::streambuf()`:

  Default (protected) constructor of the `class streambuf`.

Several protected member functions are available for input operations. The member functions marked `virtual` may or course be redefined in derived classes:

- `char *eback()`:

  `Streambuf` maintains three pointers controlling its input buffer: `eback` points to the 'end of the putback' area: characters can safely be put back up to this position. See also figure 14.4. `Eback` points to the *beginning* of the input buffer.

- `char *egptr()`:

  `Egptr` points just beyond the last character that can be retrieved from the input buffer. See also figure 14.4. If `gptr` equals `egptr` the buffer must be refilled. This should be implemented by calling `underflow`, see below.

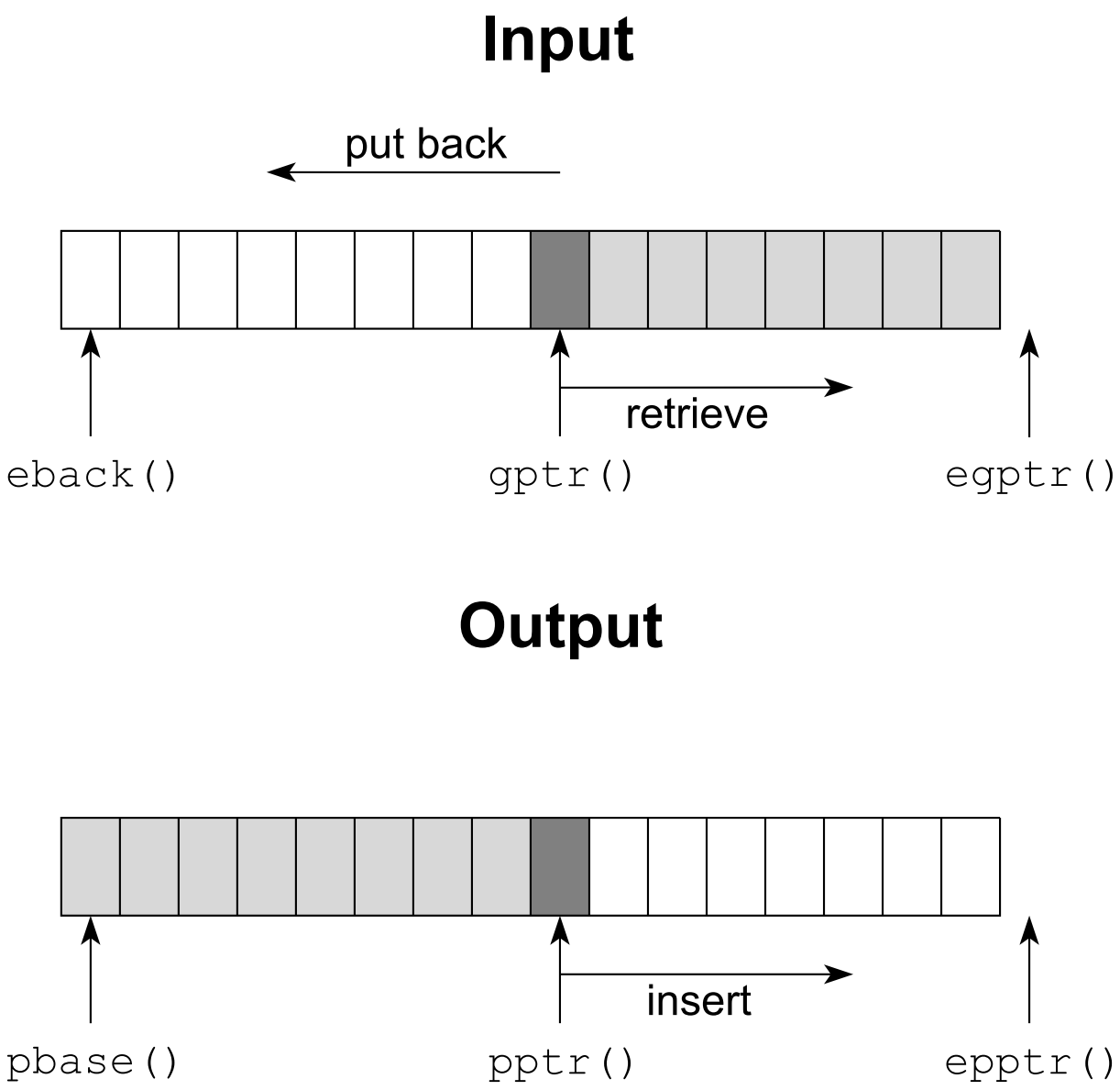

Figure 14.4: Input- and output buffer pointers of the class 'streambuf'

- `void gbump(int n)`:

  The object's `gptr` (see below) is advanced over `n` positions.

- `char *gptr()`:

  `Gptr` points to the next character to be retrieved from the object's input buffer. See also figure 14.4.

- `virtual int pbackfail(int c)`:

  This member function may be overridden by derived classes to do something intelligent when putting back character `c` fails. One might consider restoring the old read pointer when input buffer's begin has been reached. This member function is called when ungetting or putting back a character fails. In particular, it is called when

  - `gptr() == 0`: no buffering used,
  - `gptr() == eback()`: no more room to push back,
  - `*gptr() != c`: a different character than the next character to be read must be pushed back.

  If `c == endOfFile()` then the input device must be reset by one character position. Otherwise `c` must be prepended to the characters to be read. The function should return `EOF` on failure. Otherwise 0 can be returned.

- `void setg(char *beg, char *next, char *beyond)`:

  This member function initializes an input buffer. `beg` points to the beginning of the input area, `next` points to the next character to be retrieved, and `beyond` points to the location just beyond the input buffer's last character. Ususally `next` is at least `beg + 1`, to allow for a put back operation. No input buffering is used when this member is called as `setg(0, 0, 0)`. See also the member `uflow`, below.

- `virtual streamsize showmanyc()`:

  (Pronounce: s-how-many-c) This member function may be overridden by derived classes. It must return a guaranteed lower bound on the number of characters that can be read from the device before `uflow` or `underflow` returns `EOF`. By default 0 is returned (meaning no or some characters will be returned before the latter two functions return `EOF`). When a positive value is returned then the next call of `u(nder)flow` will not return `EOF`.

- `virtual int uflow()`:

  This member function may be overridden by derived classes to reload an input buffer with fresh characters. Its default implementation is to call `underflow` (see below). If `underflow()` fails, `EOF` is returned. Otherwise, the next character available character is returned as `*gptr()` following a `gbump(-1)`. `Uflow` also moves the pending character that is returned to the backup sequence. This is different from `underflow()`, which merely returns the next available character, but does not alter the input pointer positions.

  When *no* input buffering is required this function, rather than `underflow`, can be overridden to produce the next available character from the device to read from.

- `virtual int underflow()`:

  This member function may be overridden by derived classes to read another character from the device. The default implementation is to return `EOF`.

  It is called when

  - there is no input buffer (`eback() == 0`)
  - `gptr() >= egptr()`: the input buffer is exhausted.

Often, when buffering is used, the complete buffer is not refreshed as this would make it impossible to put back characters immediately following a reload. Instead, buffers are often refreshed in halves. This system is called a *split buffer*.
Classes derived from `streambuf` for reading normally at least override `underflow`. The prototypical example of an overridden `underflow` function looks like this:

```
int underflow()
{
    if (not refillTheBuffer())  // assume a member d_buffer is available
        return EOF;
                                // reset the input buffer pointers
    setg(d_buffer, d_buffer, d_buffer + d_nCharsRead);

                                // return the next available character
                                // (the cast is used to prevent
                                // misinterpretations of 0xff characters
                                // as EOF)
    return static_cast<unsigned char>(*gptr());
}
```

- `virtual streamsize xsgetn(char *buffer, streamsize n)`:

  This member function may be overridden by derived classes to retrieve at once `n` characters from the input device. The default implementation is to call `sbumpc` for every single character meaning that by default this member (eventually) calls `underflow` for every single character. The function returns the actual number of characters read or enfOfFile(). If `EOF` is returned the `streambuf` will stop reading the device.

The following protected members are available for output operations. Again, some members may be overridden by derived classes:

- `virtual int overflow(int c)`:

  This member function may be overridden by derived classes to flush the characters currently stored in the output buffer to the output device, and then to reset the output buffer pointers so as to represent an empty buffer. Its parameter `c` is initialized to the next character to be processed. If no output buffering is used `overflow` is called for every single character that is written to the `streambuf` object. No output buffering is accomplised by setting the buffer pointers (using, `setp`, see below) to 0. The default implementation returns `EOF`, indicating that no characters can be written to the device.
  Classes derived from `streambuf` for writing normally at least override `overflow`. The prototypical example of an overridden `overflow` function looks like this:

```
int OFdStreambuf::overflow(int c)
{
    sync();                             // flush the buffer
    if (c != EOF)                       // write a character?
    {
        *pptr() = static_cast<char>(c); // put it into the buffer
        pbump(1);                       // advance the buffer's pointer
    }
    return c;
}
```

- `char *pbase()`:

  `Streambuf` maintains three pointers controlling its output buffer: `pbase` points to the beginning of the output buffer area. See also figure 14.4.

- `char *epptr()`:

  `Streambuf` maintains three pointers controlling its output buffer: `epptr` points just beyond the output buffer's last available location. See also figure 14.4. If `pptr` (see below)

equals `epptr` the buffer must be flushed. This is implemented by calling `overflow`, see before.

- `void pbump(int n)`:

  The location returned by `pptr` (see below) is advanced by `n`. The next character to write will be entered at that location.

- `char *pptr()`:

  `Streambuf` maintains three pointers controlling its output buffer: `pptr` points to the location in the output buffer where the next available character should be written. See also figure 14.4.

- `void setp(char *beg, char *beyond)`:

  `Streambuf`'s output buffer is initialized to the locations passed to `setp`. `Beg` points to the beginning of the output buffer and `beyond` points just beyond the last available location of the output buffer. Use `setp(0, 0)` to indicate that `i(no buffering)` should be used. In that case `tt(overflow)` is called for every single character to write to the device.

  `streamsize xsputn(char const *buffer, streamsize n)`:

  This member function may be overridden by derived classes to write a series of at most `n` characters to the output buffer. The actual number of inserted characters is returned. If `EOF` is returned writing to the device stops. The default implementation calls `sputc` for each individual character, so redefining this member is only necessary if a more efficient implementation is required.

Several protected members are related to buffer management and positioning:

- `virtual streambuf *setbuf(char *buffer, streamsize n)`:

  This member function may be overridden by derived classes to install a buffer. The default implementation is to do nothing. It is called by `pubsetbuf`.

- `virtual pos_type seekoff(off_type offset, ios::seekdir way, ios::openmode mode = ios::in |ios::out)`

  This member function may be overridden by derived classes to reset the next pointer for input or output to a new relative position (using `ios::beg`, `ios::cur` or `ios::end`). The default implementation indicates failure by returning -1. The function is called when `tellg` or `tellp` are called. When derived class supports seeking, then it should also define this function to handle repositioning requests. It is called by `pubseekoff`. The new position or an invalid position (i.e., -1) is returned.

- `virtual pos_type seekpos(pos_type offset, ios::openmode mode = ios::in |ios::out)`:

  This member function may be overridden by derived classes to reset the next pointer for input or output to a new absolute position (i.e, relative to `ios::beg`). The default implementation indicates failure by returning -1.

- `virtual int sync()`:

  This member function may be overridden by derived classes to flush the output buffer to the output device or to reset the input device just betond the position of the character that was returned last. It returns 0 on success, -1 on failure. The default implementation (not using a buffer) is to return 0, indicating successful syncing. This member is used to ensure that any characters that are still buffered are written to the device or to put unconsumed characters back to the device when the `streambuf` object ceases to exist.

When classes are derived from `streambuf` at least `underflow` should be overridden by classes intending to read information from devices, and `overflow` should be overridden by classes intending to write information to devices. Several examples of classes derived from `streambuf` are provided in chapter 23.

`Fstream` class type objects use a combined input/output buffer. This is a result from that `istream` and `ostream` being virtually derived from `ios`, which class contains the `streambuf`. To construct a class supporting both input and output using separate buffers, the `streambuf` itself may define two buffers. When `seekoff` is called for reading, a `mode` parameter can be set to `ios::in`, otherwise to `ios::out`. Thus the derived class knows whether it should access the `read` buffer or the `write` buffer. Of course, `underflow` and `overflow` do not have to inspect the mode flag as they by implication know on which buffer they should operate.

14.7.2 The class 'filebuf'

The `class filebuf` is a specialization of `streambuf` used by the file `stream` classes. Before using a `filebuf` the header file `<fstream>` must have been included.

In addition to the (public) members that are available through the `class streambuf`, `filebuf` offers the following (public) members:

- `filebuf()`:

  `Filebuf` offers a public constructor. It initializes a plain `filebuf` object that is not yet connected to a stream.

- `bool is_open()`:

  `True` is returned if the `filebuf` is actually connected to an open file, `false` otherwise. See the `open` member, below.

- `filebuf *open(char const *name, ios::openmode mode)`:

  This member function associates the `filebuf` object with a file whose name is provided. The file is opened according to the provided `openmode`.

- `filebuf *close()`:

  This member function closes the association between the `filebuf` object and its file. The association is automatically closed when the `filebuf` object ceases to exist.

14.8 A polymorphic exception class

Earlier in the **C++** Annotations (section 9.3.1) we hinted at the possibility of designing a class `Exception` whose `process` member would behave differently, depending on the kind of exception that was thrown. Now that we've introduced polymorphism we can further develop this example.

It will probably not come as a surprise that our class `Exception` should be a polymorphic base class from which special exception handling classes can be derived. In section 9.3.1 a member `severity` was used offering functionality that may be replaced by members of the `Exception` base class.

The base class `Exception` may be designed as follows:

```
#ifndef INCLUDED_EXCEPTION_H_
#define INCLUDED_EXCEPTION_H_
#include <iostream>
#include <string>

class Exception
{
    std::string d_reason;

    public:
```

```
            Exception(std::string const &reason);
            virtual ~Exception();

            std::ostream &insertInto(std::ostream &out) const;
            void handle() const;

        private:
            virtual void action() const;
    };

    inline void Exception::action() const
    {
        throw;
    }
    inline Exception::Exception(std::string const &reason)
    :
        d_reason(reason)
    {}
    inline void Exception::handle() const
    {
        action();
    }
    inline std::ostream &Exception::insertInto(std::ostream &out) const
    {
        return out << d_reason;
    }

    inline std::ostream &operator<<(std::ostream &out, Exception const &e)
    {
        return e.insertInto(out);
    }

    #endif
```

Objects of this class may be inserted into `ostream`s but the core element of this class is the virtual member function `action`, by default rethrowing an exception.

A derived class `Warning` simply prefixes the thrown warning text by the text `Warning:`, but a derived class `Fatal` overrides `Exception::action` by calling `std::terminate`, forcefully terminating the program.

Here are the classes `Warning` and `Fatal`

```
    #ifndef WARNINGEXCEPTION_H_
    #define WARNINGEXCEPTION_H_

    #include "exception.h"

    class Warning: public Exception
    {
        public:
            Warning(std::string const &reason)
            :
                Exception("Warning: " + reason)
            {}
    };
    #endif


    #ifndef FATAL_H_
```

```
    #define FATAL_H_

    #include "exception.h"

    class Fatal: public Exception
    {
        public:
            Fatal(std::string  const &reason);
        private:
            virtual void action() const;
    };

    inline Fatal::Fatal(std::string  const &reason)
    :
        Exception(reason)
    {}

    inline void Fatal::action() const
    {
        std::cout << "Fatal::action() terminates" << std::endl;
        std::terminate();
    }

    #endif
```

When the example program is started without arguments it will throw a `Fatal` exception, otherwise it will throw a `Warning` exception. Of course, additional exception types could also easily be defined:

```
    #include "warning.h"
    #include "fatal.h"

    Exception::~Exception()
    {}

    using namespace std;

    int main(int argc, char **argv)
    try
    {
        try
        {
            if (argc == 1)
                throw Fatal("Missing Argument") ;
            else
                throw Warning("the argument is ignored");
        }
        catch (Exception const &e)
        {
            cout << e << '\n';
            e.handle();
        }
    }
    catch(...)
    {
        cout << "caught rethrown exception\n";
    }

//
//
```

```
//    void initialExceptionHandler(Exception const *e)
//    {
//        cout << *e << endl;         // show the plain-text information
//
//        if
//        (
//            !dynamic_cast<MessageException const *>(e)
//            &&
//            !dynamic_cast<WarningException const *>(e)
//        )
//            throw;                  // Pass on other types of Exceptions
//
//        e->process();               // Process a message or a warning
//        delete e;
//    }
```

14.9 How polymorphism is implemented

This section briefly describes how polymorphism is implemented in **C++**. It is not necessary to understand how polymorphism is implemented if you just want to *use* polymorphism. However, we think it's nice to know how polymorphism is possible. Also, knowing how polymorphism is implemented clarifies why there is a (small) penalty to using polymorphism in terms of memory usage and efficiency.

The fundamental idea behind polymorphism is that the compiler does not know which function to call compile-time. The appropriate function will be selected run-time. That means that the address of the function must be available somewhere, to be looked up prior to the actual call. This 'somewhere' place must be accessible to the object in question. So when a `Vehicle *vp` points to a `Truck` object, then `vp->weight()` calls `Truck`'s member function. the address of this function is obtained through the actual object to which `vp` points.

Polymorphism is commonly implemented as follows: an object containing virtual member functions also contains, usually as its first data member a hidden data member, pointing to an array containing the addresses of the class's virtual member functions. The hidden data member is usually called the *vpointer*, the array of virtual member function addresses the *vtable*.

The class's vtable is shared by all objects of that class. The overhead of polymorphism in terms of memory consumption is therefore:

- one vpointer data member per object pointing to:
- one vtable per class.

Consequently, a statement like `vp->weight` first inspects the hidden data member of the object pointed to by `vp`. In the case of the vehicle classification system, this data member points to a table containing two addresses: one pointer to the function `weight` and one pointer to the function `setWeight` (three pointers if the class also defines (as it should) a virtual destructor). The actually called function is determined from this table.

The internal organization of the objects having virtual functions is illustrated in figures Figure 14.5 and Figure 14.6 (originals provided by `Guillaume Caumon`[3]).

As shown by figures Figure 14.5 and Figure 14.6, objects potentially using virtual member functions must have one (hidden) data member to address a table of function pointers. The objects of the classes `Vehicle` and `Auto` both address the same table. The class `Truck`, however, overrides `weight`. Consequently, `Truck` needs its own vtable.

A small complication arises when a class is derived from multiple base classes, each defining virtual functions. Consider the following example:

[3] mailto:Guillaume.Caumon@ensg.inpl-nancy.fr

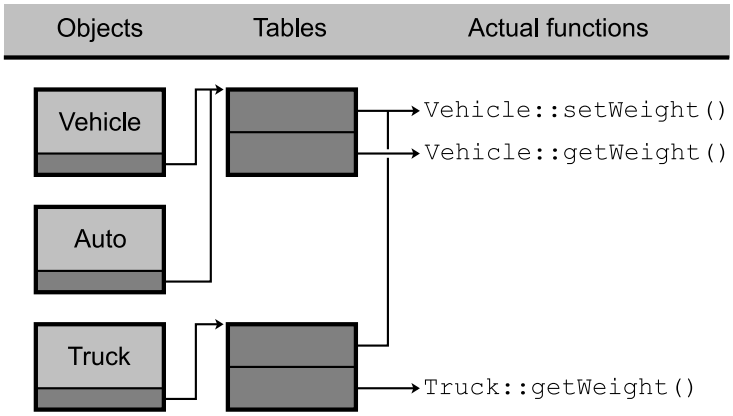

Figure 14.5: Internal organization objects when virtual functions are defined.

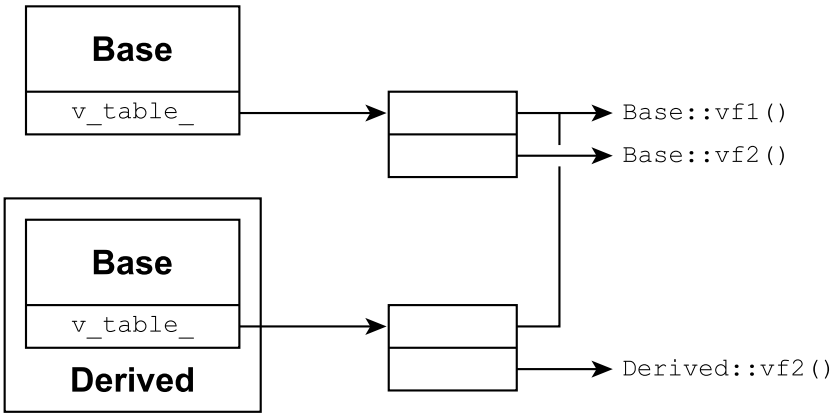

Figure 14.6: Complementary figure, provided by Guillaume Caumon

```
class Base1
{
    public:
        virtual ~Base1();
        void fun1();        // calls vOne and vTwo
    private:
        virtual void vOne();
        virtual void vTwo();
};
class Base2
{
    public:
        virtual ~Base2();
        void fun2();        // calls vThree
    private:
        virtual void vThree();
};
class Derived: public Base1, public Base2
{
    public:
        virtual ~Derived();
    private:
        virtual ~vOne();
        virtual ~vThree();
};
```

In the example `Derived` is multiply derived from `Base1` and `Base2`, each supporting virtual functions. Because of this, `Derived` also has virtual functions, and so `Derived` has a `vtable` allowing a base class pointer or reference to access the proper virtual member.

When `Derived::fun1` is called (or a `Base1` pointer pointing to `fun1` calls `fun1`) `fun1` will call `Derived::vOne` and `Base1::vTwo`. Likewise, when `Derived::fun2` is called `Derived::vThree` will be called.

The complication occurs with `Derived`'s vtable. When `fun1` is called its class type determines the vtable to use and hence which virtual member to call. So when `vOne` is called from `fun1`, it is presumably the second entry in `Derived`'s vtable, as it must match the second entry in `Base1`'s vtable. However, when `fun2` calls `vThree` it apparently is also the second entry in `Derived`'s vtable as it must match the second entry in `Base2`'s vtable.

Of course this cannot be realized by a single vtable. Therefore, when multiple inheritance is used (each base class defining virtual members) another approach is followed to determine which virtual function to call. In this situation (cf. figure Figure 14.7) the class `Derived` receives *two* `vtables`, one for each of its base classes and each `Derived` class object harbors *two* hidden vpointers, each one pointing to its corresponding vtable.

Since base class pointers, base class references, or base class interface members unambiguously refer to one of the base classes the compiler can determine which vpointer to use.

The following therefore holds true for classes multiply derived from base classes offering virtual member functions:

- the derived class defines a vtable for each of its base classes offering virtual members;
- Each derived class object contains as many hidden vpointers as it has vtables.
- Each of a derived class object's vpointers points to a unique vtable and the vpointer to use is determined by the class type of the base class pointer, the base class reference, or the base class interface function that is used.

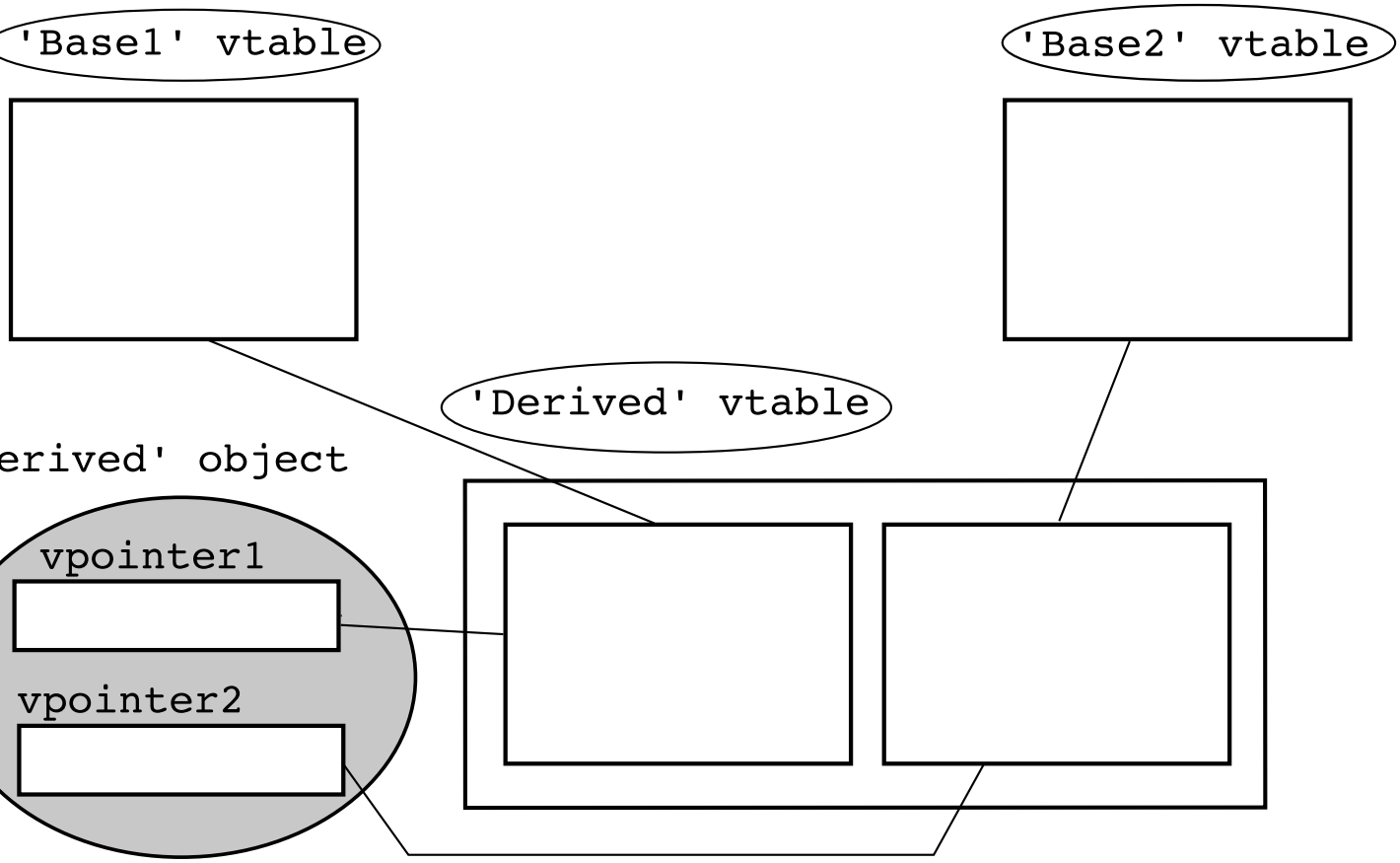

Figure 14.7: Vtables and vpointers with multiple base classes

14.10 Undefined reference to vtable ...

Occasionaly, the linker will generate an error like the following:

```
In function 'Derived::Derived()':
    : undefined reference to 'vtable for Derived'
```

This error is generated when a virtual function's implementation is missing in a derived class, but the function is mentioned in the derived class's interface.

Such a situation is easily encountered:

- Construct a (complete) base class defining a virtual member function;
- Construct a Derived class mentioning the virtual function in its interface;
- The Derived class's virtual function is not implemented. Of course, the compiler doesn't know that the derived class's function is not implemented and will, when asked, generate code to create a derived class object;
- Eventually, the linker is unable to find the derived class's virtual member function. Therefore, it is unable to construct the derived class's vtable;
- The linker complains with the message:

  ```
  undefined reference to 'vtable for Derived'
  ```

Here is an example producing the error:

```
class Base
{
    virtual void member();
};
inline void Base::member()
{}
class Derived: public Base
{
    virtual void member();      // only declared
};
int main()
{
    Derived d;  // Will compile, since all members were declared.
                // Linking will fail, since we don't have the
                // implementation of Derived::member()
}
```

It's of course easy to correct the error: implement the derived class's missing virtual member function.

14.11 Virtual constructors

In section 14.2 we learned that **C++** supports *virtual destructors*. Like many other object oriented languages (e.g., **Java**), however, the notion of a *virtual constructor* is not supported. Not having virtual constructors becomes a liability when only base class references or pointers are available, and a copy of a derived class object is required. *Gamma et al.* (1995) discuss the *Prototype design pattern* to deal with this situation.

According to the *Prototype Design Pattern* each derived class is given the responsibility of implementing a member function returning a pointer to a copy of the object for which the member is called. The usual name for this function is `clone`. Separating the user interface from the reimplementation interface `clone` is made part of the interface and `newCopy` is defined in the reimplementation interface. A base class supporting 'cloning' defines a virtual destructor, `clone`, returning `newCopy`'s return value and the *virtual copy constructor*, a pure virtual function, having the prototype `virtual Base *newCopy() const = 0`. As `newCopy` is a pure virtual function all derived classes must now implement their own 'virtual constructor'.

This setup suffices in most situations where we have a pointer or reference to a base class, but it will fail when used with abstract containers. We can't create a `vector<Base>`, with `Base` featuring the pure virtual `copy` member in its interface, as `Base` is called to initialize new elements of such a vector. This is impossible as `newCopy` is a pure virtual function, so a `Base` object can't be constructed.

The intuitive solution, providing `newCopy` with a default implementation, defining it as an ordinary virtual function, fails too as the container calls `Base(Base const &other)`, which would have to call `newCopy` to copy `other`. At this point it is unclear what to do with that copy, as the new `Base` object already exists, and contains no `Base` pointer or reference data member to assign `newCopy`'s return value to.

Alternatively (and preferred) the original `Base` class (defined as an abstract base class) is kept as-is and a wrapper class `Clonable` is used to manage the `Base` class pointers returned by `newCopy`. In chapter 17 ways to merge `Base` and `Clonable` into one class are discussed, but for now we'll define `Base` and `Clonable` as separate classes.

The class `Clonable` is a very standard class. It contains a pointer member so it needs a copy constructor, destructor, and overloaded assignment operator. It's given at least one non-standard member: `Base &base() const`, returning a reference to the derived object to which `Clonable`'s `Base *` data member refers. It is also privided with an additional constructor to initialize its `Base *` data member.

Any non-abstract class derived from `Base` must implement `Base *newCopy()`, returning a pointer to a newly created (allocated) copy of the object for which `newCopy` is called.

Once we have defined a derived class (e.g., `Derived1`), we can put our `Clonable` and `Base` facilities to good use. In the next example we see `main` defining a `vector<Clonable>`. An anonymous `Derived1` object is then inserted into the vector using the following steps:

- A new anonymous `Derived1` object is created;
- It initializes a `Clonable` using `Clonable(Base *bp)`;
- The just created `Clonable` object is inserted into the vector, using `Clonable`'s move constructor. There are only temporary `Derived` and `Clonable` objects at this point, so no copy construction is required.

In this sequence, only the `Clonable` object containing the `Derived1 *` is used. No additional copies need to be made (or destroyed).

Next, the `base` member is used in combination with `typeid` to show the actual type of the `Base &` object: a `Derived1` object.

`Main` then contains the interesting definition `vector<Clonable> v2(bv)`. Here a copy of `bv` is created. This copy construction observes the actual types of the `Base` references, making sure that the appropriate types appear in the vector's copy.

At the end of the program, we have created two `Derived1` objects, which are correctly deleted by the vector's destructors. Here is the full program, illustrating the 'virtual constructor' concept[4]:

```
#include <iostream>
```

[4] Jesse van den Kieboom created an alternative implementation of a class `Clonable`, implemented as a class template. His implementation is found `here`[5].

```
    #include <vector>
    #include <algorithm>
    #include <typeinfo>

// Base and its inline member:
    class Base
    {
        public:
            virtual ~Base();
            Base *clone() const;
        private:
            virtual Base *newCopy() const = 0;
    };
    inline Base *Base::clone() const
    {
        return newCopy();
    }

// Clonable and its inline members:
    class Clonable
    {
        Base *d_bp;

        public:
            Clonable();
            explicit Clonable(Base *base);
            ~Clonable();
            Clonable(Clonable const &other);
            Clonable(Clonable const &&tmp);
            Clonable &operator=(Clonable const &other);
            Clonable &operator=(Clonable const &&tmp);

            Base &base() const;
    };
    inline Clonable::Clonable()
    :
        d_bp(0)
    {}
    inline Clonable::Clonable(Base *bp)
    :
        d_bp(bp)
    {}
    inline Clonable::Clonable(Clonable const &other)
    :
        d_bp(other.d_bp->clone())
    {}
    inline Clonable::Clonable(Clonable const &&tmp)
    :
        d_bp(tmp.d_bp)
    {
        const_cast<Clonable &>(tmp).d_bp = 0;
    }
    inline Clonable::~Clonable()
    {
        delete d_bp;
    }
    inline Base &Clonable::base() const
    {
        return *d_bp;
```

```
    }

// Derived and its inline member:
    class Derived1: public Base
    {
        public:
            ~Derived1();
        private:
            virtual Base *newCopy() const;
    };
    inline Base *Derived1::newCopy() const
    {
        return new Derived1(*this);
    }

// Members not implemented inline:
    Base::~Base()
    {}
    Clonable &Clonable::operator=(Clonable const &other)
    {
        Clonable tmp(other);
        std::swap(d_bp, tmp.d_bp);
        return *this;
    }
    Clonable &Clonable::operator=(Clonable const &&tmp)
    {
        std::swap(d_bp, const_cast<Clonable &>(tmp).d_bp);
        return *this;
    }
    Derived1::~Derived1()
    {
        std::cout << "~Derived1() called\n";
    }

// The main function:
    using namespace std;

    int main()
    {
        vector<Clonable> bv;

        bv.push_back(Clonable(new Derived1()));
        cout << "bv[0].name: " << typeid(bv[0].base()).name() << '\n';

        vector<Clonable> v2(bv);
        cout << "v2[0].name: " << typeid(v2[0].base()).name() << '\n';
    }
    /*
        Output:
            bv[0].name: 8Derived1
            v2[0].name: 8Derived1
            ~Derived1() called
            ~Derived1() called
    */
```

Chapter 15

Friends

In all examples discussed up to now, we've seen that `private` members are only accessible by the members of their class. This is *good*, as it enforces encapsulation and data hiding: By encapsulating functionality in classes we prevent that classes have multiple responsibilities; by hiding its data from external code we promote a class's data integrity and we prevent that external code becomes implementation dependent on the data in a class.

In this (very) short chapter we introduce the `friend` keyword and the principles that underly its use. The bottom line being that by using the `friend` keyword functions are granted access to a class's `private` members. As will be discussed this does not automatically imply that we abandon the principle of data hiding when the `friend` keyword is used.

In this chapter the topic of friendship among classes is not discussed. Situations in which it is natural to use friendship among classes are discussed in chapters 17 and 20 and are a natural extension of the way friendship is handled for functions.

There should be a well-defined conceptual reason for declaring friendship (i.e., using the `friend` keyword). The traditionally offered definition of the class concept usually looks something like this:

> *A class is a set of data together with the functions that operate on that set of data.*

As we've seen in chapter 10 some functions have to be defined outside of a class interface. They are defined outside of the class interface to allow promotions for their operands or to extend the facilities of existing classes not directly under our control. According to the above traditional definition of the class concept those functions that cannot be defined in the class interface itself should nevertheless be considered functions belonging to the class. Stated otherwise: if permitted by the language's syntax they would certainly have been defined inside the class interface. There are two ways to implement such functions. One way consists of implementing those functions using available public member functions. This approach was used, e.g., in section 10.2. Another approach applies the definition of the class concept to those functions. By stating that those functions in fact belong to the class they should be given direct access to the data members of objects. This is accomplished by the `friend` keyword.

A a general principle we state that all functions operating on the data of objects of a class that are declared in the same file as the class interface itself belong to that class and may be granted direct access to the class's data members.

15.1 Friend functions

In section **??** the insertion operator of the class `Person` (cf. section 8.3) was implemented like this:

```
ostream &operator<<(ostream &out, Person const &person)
```

```
{
    return
        out <<
            "Name:    " << person.name() <<  ", "
            "Address: " << person.address() <<  ", "
            "Phone:   " << person.phone();
}
```

`Person` objects can now be inserted into streams.

However, this implementation required three member functions to be called, which may be considered a source of inefficiency. An improvement would be reached by defining a member `Person::insertInto` and let `operator«` call that function. These two functions could be defined as follows:

```
std::ostream &operator<<(std::ostream &out, Person const &person)
{
    return person.insertInto(out);
}
std::ostream &Person::insertInto(std::ostream &out)
{
    return
        cout << "Name:    " << d_name << ", "
                "Address: " << d_address << ", "
                "Phone:   " << d_phone;
}
```

As `insertInto` is a member function it has direct access to the object's data members so no additional member functions must be called when inserting `person` into `out`.

The next step consists of realizing that `insertInto` is only defined for the benefit of `operator«`, and that `operator«`, as it is declared in the header file containing `Person`'s class interface should be considered a function belonging to the class `Person`. The member `insertInto` can therefore be omitted when `operator«` is declared as a friend.

Friend functions must be declared as friends in the class interface. These *friend declarations* are not *member* functions, and so they are independent of the class's `private, protected` and `public` sections. Friend declaration may be placed anywhere in the class interface. Convention dictates that friend declaractions are listed directly at the top of the class interface. The class `Person`, using `friend` declaration for its extraction and insertion operators starts like this:

```
class Person
{
    friend std::ostream &operator<<(std::ostream &out, Person &pd);
    friend std::istream &operator>>(std::istream &in, Person &pd);

    // previously shown interface (data and functions)
};
```

The insertion operator may now directly access a `Person` object's data members:

```
std::ostream &operator<<(std::ostream &out, Person const &person)
{
    return
        cout << "Name:    " << person.d_name << ", "
                "Address: " << person.d_address << ", "
                "Phone:   " << person.d_phone;
}
```

Friend declarations are true declarations. Once a class contains friend declarations these friend functions do not have to be declared again below the class's interface. This also clearly indicates the class designer's intent: the friend functions are declared by the class, and can thus be considered functions belonging to the class.

Chapter 16

Classes Having Pointers To Members

Classes having `pointer data members` have been discussed in detail in chapter 8. Classes defining pointer data-members deserve some special attention, as they usually require the definitions of copy constructors, overloaded assignment operators and destructors

Situations exist where we do not need a pointer to an object but rather a pointer to members of a class. Pointers to members can profitably be used to configure the behavior of objects of classes. Depending on which member a pointer to a member points to objects will show certain behavior.

Although pointers to members have their use, polymorphism can frequently be used to realize comparable behavior. Consider a class having a member `process` performing one of a series of alternate behaviors. Instead of selecting the behavior of choice at object construction time the class could use the interface of some (abstract) base class, passing an object of some derived class to its constructor and could thus configure its behavior. This allows for easy, extensible and flexible configuration, but access to the class's data members would be less flexible and would possibly require the use of 'friend' declarations. In such cases pointers to members may actually be preferred as this allows for (somewhat less flexible) configuration as well as direct access to a class's data members.

So the choice apparently is between on the one hand ease of configuration and on the other hand ease of access to a class's data members. In this chapter we'll concentrate on pointers to members, investigating what these pointers have to offer.

16.1 Pointers to members: an example

Knowing how pointers to variables and objects are used does not intuitively lead to the concept of *pointers to members* . Even if the return types and parameter types of member functions are taken into account, surprises can easily be encountered. For example, consider the following class:

```
class String
{
    char const *(*d_sp)() const;

    public:
        char const *get() const;
};
```

For this class, it is not possible to let `char const *(*d_sp)() const` point to the `String::get` member function as `d_sp` cannot be given the address of the member function `get`.

One of the reasons why this doesn't work is that the variable `d_sp` has global scope (it is a pointer to *a* function, not a pointer to a function within `String`), while the member function `get` is defined within the `String` class, and thus has `class scope`. The fact that `d_sp` is a data member of the class `String` is irrelevant here. According to `d_sp`'s definition, it points to a function living somewhere *outside* of the class.

Consequently, to define a pointer to a member (either data or function, but usually a function) of a class, the scope of the pointer must indicate class scope. Doing so, a pointer to the member `String::get` is defined like this:

```
char const *(String::*d_sp)() const;
```

So, by prefixing the `*d_sp` pointer data member by `String::`, it is defined as a pointer in the context of the class `String`. According to its definition it is *a pointer to a function in the class `String`, not expecting arguments, not modifying its object's data, and returning a pointer to constant characters*.

16.2 Defining pointers to members

Pointers to members are defined by prefixing the normal pointer notation with the appropriate class plus scope resolution operator. Therefore, in the previous section, we used `char const * (String::*d_sp)() const` to indicate that `d_sp`

- is a pointer (`*d_sp`);
- points to something in the class `String` (`String::*d_sp`);
- is a pointer to a `const` function, returning a `char const *` (`char const * (String::*d_sp)() const`).

The prototype of a matching function is therefore:

```
char const *String::somefun() const;
```

which is any `const` parameterless function in the class `String`, returning a `char const *`.

When defining pointers to members the standard procedure for constructing pointers to functions can still be applied:

- put parentheses around the fully qualified function name (i.e., the function's header, including the function's class name):

  ```
  char const * ( String::somefun ) () const
  ```

- Put a pointer (a star (`*`)) character immediately before the function name itself:

  ```
  char const * ( String:: * somefun ) () const
  ```

- Replace the function name with the name of the pointer variable:

  ```
  char const * (String::*d_sp)() const
  ```

Here is another example, defining a pointer to a data member. Assume the class `String` contains a `string d_text` member. How to construct a pointer to this member? Again we follow standard procedure:

- put parentheses around the fully qualified variable name:

  ```
  std::string (String::d_text)
  ```

- Put a pointer (a star (`*`)) character immediately before the variable-name itself:

```
std::string (String::*d_text)
```

- Replace the variable name with the name of the pointer variable:

```
std::string (String::*tp)
```

  In this case, the parentheses are superfluous and may be omitted:

```
string String::*tp
```

Alternatively, a very simple rule of thumb is

- Define a normal (i.e., global) pointer variable,
- Prefix the class name to the pointer character, once you point to something inside a class

For example, the following pointer to a global function

```
char const * (*sp)() const;
```

becomes a pointer to a member function after prefixing the class-scope:

```
char const * (String::*sp)() const;
```

Nothing forces us to define pointers to members in their target (`String`) classes. Pointers to members *may* be defined in its target class (so it becomes a data member), or in another class, or as a local variable or as a global variable. In all these cases the pointer to member variable can be given the address of the kind of member it points to. The important part is that a pointer to member can be initialized or assigned without requiring the existence an object of the pointer's target class.

Initializing or assigning an address to such a pointer merely indicates to which member the pointer points. This can be considered some kind of *relative address*; relative to the object for which the function is called. No object is required when pointers to members are initialized or assigned. While it is allowed to initialize or assign a pointer to member, it is (of course) not possible to *call* those members without specifying an object of the correct type.

In the following example initialization of and assignment to pointers to members is illustrated (for illustration purposes all members of the class `PointerDemo` are defined `public`). In the example itself the `&`-operator is used to determine the addresses of the members. These operators as well as the class-scopes are required. Even when used inside member implementations:

```
class PointerDemo
{
    public:
        size_t d_value;
        size_t get() const;
};

inline size_t PointerDemo::get() const
{
    return d_value;
}

int main()
{                                           // initialization
    size_t (PointerDemo::*getPtr)() const = &PointerDemo::get;
```

```
    size_t PointerDemo::*valuePtr         = &PointerDemo::d_value;

    getPtr   = &PointerDemo::get;           // assignment
    valuePtr = &PointerDemo::d_value;
}
```

This involves nothing special. The difference with pointers at global scope is that we're now restricting ourselves to the scope of the `PointerDemo` class. Because of this restriction, all *pointer* definitions and all variables whose addresses are used must be given the `PointerDemo` class scope.

Pointers to members can also be used with `virtual` member functions. No special syntax is required when pointing to virtual members. Pointer construction, initialization and assignment is done identically to the way it is done with non-virtual members.

16.3 Using pointers to members

Using pointers to members to call a member function requires the existence of an object of the class of the members to which the pointer to member refers to. With pointers operating at global scope, the dereferencing operator `*` is used. With pointers to objects the field selector operator operating on pointers (`->`) or the field selector operating operating on objects (`.`) can be used to select appropriate members.

To use a pointer to member in combination with an object the pointer to member field selector (`.*`) must be specified. To use a pointer to a member via a pointer to an object the 'pointer to member field selector through a pointer to an object' (`->*`) must be specified. These two operators combine the notions of a field selection (the `.` and `->` parts) to reach the appropriate field in an object and of dereferencing: a dereference operation is used to reach the function or variable the pointer to member points to.

Using the example from the previous section, let's see how we can use pointers to members function and pointers to data members:

```
#include <iostream>

class PointerDemo
{
    public:
        size_t d_value;
        size_t get() const;
};

inline size_t PointerDemo::get() const
{
    return d_value;
}

using namespace std;

int main()
{                                           // initialization
    size_t (PointerDemo::*getPtr)() const = &PointerDemo::get;
    size_t PointerDemo::*valuePtr   = &PointerDemo::d_value;

    PointerDemo object;                     // (1) (see text)
    PointerDemo *ptr = &object;

    object.*valuePtr = 12345;               // (2)
    cout << object.*valuePtr << '\n' <<
```

```
            object.d_value << '\n';

    ptr->*valuePtr = 54321;                 // (3)
    cout << object.d_value << '\n' <<
            (object.*getPtr)() << '\n' <<   // (4)
            (ptr->*getPtr)() << '\n';
}
```

We note:

- At (1) a `PointerDemo` object and a pointer to such an object is defined.
- At (2) we specify an object (and hence the `.*` operator) to reach the member `valuePtr` points to. This member is given a value.
- At (3) the same member is assigned another value, but this time using the pointer to a `PointerDemo` object. Hence we use the `->*` operator.
- At (4) the `.*` and `->*` are used once again, this time to call a function through a pointer to member. As the function argument list has a higher priority than the pointer to member field selector operator, the latter *must* be protected by parentheses.

Pointers to members can be used profitably in situations where a class has a member that behaves differently depending on a configuration setting. Consider once again the class `Person` from section 8.3. `Person` defines data members holding a person's name, address and phone number. Assume we want to construct a `Person` database of employees. The employee database can be queried, but depending on the kind of person querying the database either the name, the name and phone number or all stored information about the person is made available. This implies that a member function like `address` must return something like '`<not available>`' in cases where the person querying the database is not allowed to see the person's address, and the actual address in other cases.

The employee database is opened specifying an argument reflecting the status of the employee who wants to make some queries. The status could reflect his or her position in the organization, like `BOARD`, `SUPERVISOR`, `SALESPERSON`, or `CLERK`. The first two categories are allowed to see all information about the employees, a `SALESPERSON` is allowed to see the employee's phone numbers, while the `CLERK` is only allowed to verify whether a person is actually a member of the organization.

We now construct a member `string personInfo(char const *name)` in the database class. A standard implementation of this class could be:

```
string PersonData::personInfo(char const *name)
{
    Person *p = lookup(name);   // see if 'name' exists

    if (!p)
        return "not found";

    switch (d_category)
    {
        case BOARD:
        case SUPERVISOR:
            return allInfo(p);
        case SALESPERSON:
            return noPhone(p);
        case CLERK:
            return nameOnly(p);
    }
}
```

Although it doesn't take much time, the `switch` must nonetheless be evaluated every time `personInfo()` is called. Instead of using a switch, we could define a member `d_infoPtr` as a pointer to a member function of the class `PersonData` returning a `string` and expecting a `Person` reference as its argument.

Instead of evaluating the switch this pointer can be used to point to `allInfo`, `noPhone` or `nameOnly`. Furthermore, the member function the pointer points to will be known by the time the `PersonData` object is constructed and so its value needs to be determined only once (at the PersonData) object's construction time.

Having initialized `d_infoPtr` `personInfo()` member function is now implemented simply as:

```
string PersonData::personInfo(char const *name)
{
    Person *p = lookup(name);       // see if 'name' exists

    return p ? (this->*d_infoPtr)(p) :  "not found";
}
```

The member `d_infoPtr` is defined as follows (within the class `PersonData`, omitting other members):

```
class PersonData
{
    string (PersonData::*d_infoPtr)(Person *p);
};
```

Finally, the constructor initializes `d_infoPtr`. This could be realized using a simple switch:

```
PersonData::PersonData(PersonData::EmployeeCategory cat)
:
    switch (cat)
    {
        case BOARD:
        case SUPERVISOR:
            d_infoPtr = &PersonData::allInfo;
        break;
        case SALESPERSON:
            d_infoPtr = &PersonData::noPhone;
        break;
        case CLERK:
            d_infoPtr = &PersonData::nameOnly;
        break;
    }
}
```

Note how addresses of member functions are determined. The class `PersonData` scope *must* be specified, even though we're already inside a member function of the class `PersonData`.

An example using pointers to data members is provided in section 19.1.60, in the context of the `stable_sort` generic algorithm.

16.4 Pointers to static members

Static members of a class can be used without having available an object of their class. Public static members can be called like free functions, albeit that their class names must be specified when they are called.

Assume a class `String` has a public static member function `count`, returning the number of string objects created so far. Then, without using any `String` object the function `String::count` may be called:

```
void fun()
{
    cout << String::count() << '\n';
}
```

Public static members can be called like free functions (but see also section 11.2.1). *Private* static members can only be called within the context of their class, by their class's member or friend functions.

Since static members have no associated objects their addresses can be stored in ordinary function pointer variables, operating at the global level. Pointers to members cannot be used to store addresses of static members. Example:

```
void fun()
{
    size_t (*pf)() = String::count;
            // initialize pf with the address of a static member function

    cout << (*pf)() << '\n';
            // displays the value returned by String::count()
}
```

16.5 Pointer sizes

An interesting characteristic of pointers to members is that their sizes differ from those of 'normal' pointers. Consider the following little program:

```
#include <string>
#include <iostream>

class X
{
    public:
        void fun();
        std::string d_str;
};
inline void X::fun()
{
    std::cout << "hello\n";
}

using namespace std;
int main()
{
    cout <<
       "size of pointer to data-member:     " << sizeof(&X::d_str) << "\n"
       "size of pointer to member function: " << sizeof(&X::fun) << "\n"
       "size of pointer to non-member data: " << sizeof(char *) << "\n"
       "size of pointer to free function:   " << sizeof(&printf) << '\n';
}

/*
    generated output:
```

```
        size of pointer to data-member:     4
        size of pointer to member function: 8
        size of pointer to non-member data: 4
        size of pointer to free function:   4
    */
```

On a 32-bit architecture a pointer to a member function requires eight bytes, whereas other kind of pointers require four bytes (Using Gnu's g++ compiler).

Pointer sizes are hardly ever explicitly used, but their sizes may cause confusion in statements like:

```
printf("%p", &X::fun);
```

Of course, `printf` is likely not the right tool to produce the value of these **C++** specific pointers. The values of these pointers can be inserted into streams when a `union`, reinterpreting the 8-byte pointers as a series of size_t `char` values, is used:

```
#include <string>
#include <iostream>
#include <iomanip>

class X
{
    public:
        void fun();
        std::string d_str;
};
inline void X::fun()
{
    std::cout << "hello\n";
}

using namespace std;
int main()
{
    union
    {
        void (X::*f)();
        unsigned char *cp;
    }
        u = { &X::fun };

    cout.fill('0');
    cout << hex;
    for (unsigned idx = sizeof(void (X::*)()); idx-- > 0; )
        cout << setw(2) << static_cast<unsigned>(u.cp[idx]);
    cout << '\n';
}
```

COMMENT (Release ~pre2)

Chapter 17

Nested Classes

Classes can be defined inside other classes. Classes that are defined inside other classes are called *nested classes*. Nested classes are used in situations where the nested class has a close conceptual relationship to its surrounding class. For example, with the class `string` a type `string::iterator` is available which will provide all characters that are stored in the `string`. This `string::iterator` type could be defined as an object `iterator`, defined as nested class in the class `string`.

A class can be nested in every part of the surrounding class: in the `public, protected` or `private` section. Such a nested class can be considered a member of the surrounding class. The normal access and rules in classes apply to nested classes. If a class is nested in the `public` section of a class, it is visible outside the surrounding class. If it is nested in the `protected` section it is visible in subclasses, derived from the surrounding class (see chapter 13), if it is nested in the `private` section, it is only visible for the members of the surrounding class.

The surrounding class has no special privileges with respect to the nested class. So, the nested class still has full control over the accessibility of its members by the surrounding class. For example, consider the following class definition:

```
class Surround
{
    public:
        class FirstWithin
        {
            int d_variable;

            public:
                FirstWithin();
                int var() const;
        };
    private:
        class SecondWithin
        {
            int d_variable;

            public:
                SecondWithin();
                int var() const;
        };
};
inline int Surround::FirstWithin::var() const
{
    return d_variable;
}
inline int Surround::SecondWithin::var() const
```

```
{
    return d_variable;
}
```

In this definition access to the members is defined as follows:

- The class `FirstWithin` is visible both outside and inside `Surround`. The class `FirstWithin` therefore has global scope.
- The constructor `FirstWithin()` and the member function `var()` of the class `FirstWithin` are also globally visible.
- The `int d_variable` datamember is only visible to the members of the class `FirstWithin`. Neither the members of `Surround` nor the members of `SecondWithin` can access `d_variable` of the class `FirstWithin` directly.
- The class `SecondWithin` is only visible inside `Surround`. The public members of the class `SecondWithin` can also be used by the members of the class `FirstWithin`, as nested classes can be considered members of their surrounding class.
- The constructor `SecondWithin()` and the member function `var()` of the class `SecondWithin` can also only be reached by the members of `Surround` (and by the members of its nested classes).
- The `int d_variable` datamember of the class `SecondWithin` is only visible to the members of the class `SecondWithin`. Neither the members of `Surround` nor the members of `FirstWithin` can access `d_variable` of the class `SecondWithin` directly.
- As always, an object of the class type is required before its members can be called. This also holds true for nested classes.

If the surrounding class should have access rights to the private members of its nested classes or if nested classes should have access rights to the private members of the surrounding class, the classes can be defined as `friend` classes (see section 17.3).

The nested classes can be considered members of the surrounding class, but the members of nested classes are *not* members of the surrounding class. So, a member of the class `Surround` may not access `FirstWithin::var()` directly. This is understandable considering the fact that a `Surround` object is not also a `FirstWithin` or `SecondWithin` object. In fact, nested classes are just typenames. It is not implied that objects of such classes automatically exist in the surrounding class. If a member of the surrounding class should use a (non-static) member of a nested class then the surrounding class must define a nested class object, which can thereupon be used by the members of the surrounding class to use members of the nested class.

For example, in the following class definition there is a surrounding class `Outer` and a nested class `Inner`. The class `Outer` contains a member function `caller()` which uses the `inner` object that is composed in `Outer` to call the `infunction()` member function of `Inner`:

```
class Outer
{
    public:
        void caller();

    private:
        class Inner
        {
            public:
                void infunction();
        };
        Inner d_inner;      // class Inner must be known
};
void Outer::caller()
```

```
{
    d_inner.infunction();
}
```

The mentioned function `Inner::infunction()` can be called as part of the inline definition of `Outer::caller()`, even though the definition of the class `Inner` is yet to be seen by the compiler. On the other hand, the compiler must have seen the definition of the class `Inner` before a data member of that class can be defined.

17.1 Defining nested class members

Member functions of nested classes may be defined as inline functions. Inline member functions can be defined as if they were functions defined outside of the class definition: if the function `Outer::caller()` would have been defined outside of the class `Outer`, the full class definition (including the definition of the class `Inner`) would have been available to the compiler. In that situation the function is perfectly compilable. Inline functions can be compiled accordingly: they can be defined and they can use any nested class. Even if it appears later in the class interface.

As shown, when (nested) member functions are defined inline, their definition should be put below their class interface. Static nested data members are also normally defined outside of their classes. If the class `FirstWithin` would have a `static size_t` datamember `epoch`, it could be initialized as follows:

```
size_t Surround::FirstWithin::epoch = 1970;
```

Furthermore, multiple scope resolution operators are needed to refer to public static members in code outside of the surrounding class:

```
void showEpoch()
{
    cout << Surround::FirstWithin::epoch = 1970;
}
```

Inside the members of the class `Surround` only the `FirstWithin::` scope must be used; inside the members of the class `FirstWithin` there is no need to refer explicitly to the scope.

What about the members of the class `SecondWithin`? The classes `FirstWithin` and `SecondWithin` are both nested within `Surround`, and can be considered members of the surrounding class. Since members of a class may directly refer to each other, members of the class `SecondWithin` can refer to (public) members of the class `FirstWithin`. Consequently, members of the class `SecondWithin` could refer to the `epoch` member of `FirstWithin` as

```
    FirstWithin::epoch
```

17.2 Declaring nested classes

Nested classes may be declared before they are actually defined in a surrounding class. Such forward declarations are required if a class contains multiple nested classes, and the nested classes contain pointers, references, parameters or return values to objects of the other nested classes.

For example, the following class `Outer` contains two nested classes `Inner1` and `Inner2`. The class `Inner1` contains a pointer to `Inner2` objects, and `Inner2` contains a pointer to `Inner1` objects. Such cross references require forward declarations. These forward declarations must be specified in the same

access-category as their actual definitions. In the following example the `Inner2` forward declaration must be given in a `private` section, as its definition is also part of the class `Outer`'s private interface:

```
class Outer
{
    private:
        class Inner2;       // forward declaration

        class Inner1
        {
            Inner2 *pi2;    // points to Inner2 objects
        };
        class Inner2
        {
            Inner1 *pi1;    // points to Inner1 objects
        };
};
```

17.3 Accessing private members in nested classes

To allow nested classes to access the private members of their surrounding class; to access the private members of other nested classes; or to allow the surrounding class to access the private members of its nested classes, the `friend` keyword must be used. Consider the following situation, in which a class `Surround` has two nested classes `FirstWithin` and `SecondWithin`, while each class has a static data member `int s_variable`:

```
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            static int s_variable;
            public:
                int value();
        };
        int value();
    private:
        class SecondWithin
        {
            static int s_variable;
            public:
                int value();
        };
};
```

If the class `Surround` should be able to access `FirstWithin` and `SecondWithin`'s private members, these latter two classes must declare `Surround` to be their friend. The function `Surround::value()` can thereupon access the private members of its nested classes. For example (note the `friend` declarations in the two nested classes):

```
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
```

```
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
        int value();
    private:
        class SecondWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
};
inline int Surround::FirstWithin::value()
{
    FirstWithin::s_variable = SecondWithin::s_variable;
    return (s_variable);
}
```

Now, to allow the nested classes access to the private members of their surrounding class, the class `Surround` must declare its nested classes as friends. The `friend` keyword may only be used when the class that is to become a friend is already known as a class by the compiler, so either a forward declaration of the nested classes is required, which is followed by the friend declaration, or the friend declaration follows the definition of the nested classes. The forward declaration followed by the friend declaration looks like this:

```
class Surround
{
    class FirstWithin;
    class SecondWithin;
    friend class FirstWithin;
    friend class SecondWithin;

    public:
        class FirstWithin;
    ...
```

Alternatively, the friend declaration may follow the definition of the classes. Note that a class can be declared a friend following its definition, while the inline code in the definition already uses the fact that it will be declared a friend of the outer class. When defining members within the class interface implementations of nested class members may use members of the surrounding class that have not yet been seen by the compiler. Finally note that q‘`s_variable`’ which is defined in the class `Surround` is accessed in the nested classes as `Surround::s_variable`:

```
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
```

```
            friend class FirstWithin;
            int value();

        private:
            class SecondWithin
            {
                friend class Surround;
                static int s_variable;
                public:
                    int value();
            };
            static void classMember();

            friend class SecondWithin;
    };

    inline int Surround::value()
    {
        FirstWithin::s_variable = SecondWithin::s_variable;
        return s_variable;
    }

    inline int Surround::FirstWithin::value()
    {
        Surround::s_variable = 4;
        Surround::classMember();
        return s_variable;
    }

    inline int Surround::SecondWithin::value()
    {
        Surround::s_variable = 40;
        return s_variable;
    }
```

Finally, we want to allow the nested classes access to each other's private members. Again this requires some `friend` declarations. In order to allow `FirstWithin` to access `SecondWithin`'s private members nothing but a `friend` declaration in `SecondWithin` is required. However, to allow `SecondWithin` to access the private members of `FirstWithin` the `friend class SecondWithin` declaration cannot plainly be given in the class `FirstWithin`, as the definition of `SecondWithin` is as yet unknown. A forward declaration of `SecondWithin` is required, and this forward declaration must be provided by the class `Surround`, rather than by the class `FirstWithin`.

Clearly, the forward declaration `class SecondWithin` in the class `FirstWithin` itself makes no sense, as this would refer to an external (global) class `SecondWithin`. Likewise, it is impossible to provide the forward declaration of the nested class `SecondWithin` inside `FirstWithin` as `class Surround::SecondWithin`, with the compiler issuing a message like

```
'Surround' does not have a nested type named 'SecondWithin'
```

The proper procedure here is to declare the class `SecondWithin` in the class `Surround`, before the class `FirstWithin` is defined. Using this procedure, the friend declaration of `SecondWithin` is accepted inside the definition of `FirstWithin`. The following class definition allows full access of the private members of all classes by all other classes:

```
class Surround
{
    class SecondWithin;
```

```
        static int s_variable;
        public:
            class FirstWithin
            {
                friend class Surround;
                friend class SecondWithin;
                static int s_variable;
                public:
                    int value();
            };
            friend class FirstWithin;
            int value();
        private:
            class SecondWithin
            {
                friend class Surround;
                friend class FirstWithin;
                static int s_variable;
                public:
                    int value();
            };
            friend class SecondWithin;
    };
    inline int Surround::value()
    {
        FirstWithin::s_variable = SecondWithin::s_variable;
        return s_variable;
    }

    inline int Surround::FirstWithin::value()
    {
        Surround::s_variable = SecondWithin::s_variable;
        return s_variable;
    }

    inline int Surround::SecondWithin::value()
    {
        Surround::s_variable = FirstWithin::s_variable;
        return s_variable;
    }
```

17.4 Nesting enumerations

Enumerations too may be nested in classes. Nesting enumerations is a good way to show the close connection between the enumeration and its class. Nested enumerations have the same controlled visibility as other class members. They may be defined in the private, protected or public sections of classes and are inherited over class hierarchies. In the class `ios` we've seen values like `ios::beg` and `ios::cur`. In the current Gnu **C++** implementation these values are defined as values in the `seek_dir` enumeration:

```
    class ios: public _ios_fields
    {
        public:
            enum seek_dir
            {
                beg,
                cur,
```

```
            end
        };
};
```

For illustration purposes, let's assume that a class `DataStructure` may be traversed in a forward or backward direction. Such a class can define an enumeration `Traversal` having the values `forward` and `backward`. Furthermore, a member function `setTraversal()` can be defined requiring either of the two enumeration values. The class can be defined as follows:

```
class DataStructure
{
    public:
        enum Traversal
        {
            forward,
            backward
        };
        setTraversal(Traversal mode);
    private:
        Traversal
            d_mode;
};
```

Within the class `DataStructure` the values of the `Traversal` enumeration can be used directly. For example:

```
void DataStructure::setTraversal(Traversal mode)
{
    d_mode = mode;
    switch (d_mode)
    {
        forward:
        break;

        backward:
        break;
    }
}
```

Ouside of the class `DataStructure` the name of the enumeration type is not used to refer to the values of the enumeration. Here the classname is sufficient. Only if a variable of the enumeration type is required the name of the enumeration type is needed, as illustrated by the following piece of code:

```
void fun()
{
    DataStructure::Traversal                // enum typename required
        localMode = DataStructure::forward; // enum typename not required

    DataStructure ds;
                                            // enum typename not required
    ds.setTraversal(DataStructure::backward);
}
```

Again, only if `DataStructure` defines a nested class `Nested`, in turn defining the enumeration `Traversal`, the two class scopes are required. In that case the latter example should have been coded as follows:

```
void fun()
```

```
{
    DataStructure::Nested::Traversal
        localMode = DataStructure::Nested::forward;

    DataStructure ds;

    ds.setTraversal(DataStructure::Nested::backward);
}
```

17.4.1 Empty enumerations

`Enum` types usually have values. However, this is not required. In section 14.5.1 the `std::bad_cast` type was introduced. A `std::bad_cast` is thrown by the `dynamic_cast<>()` operator when a reference to a base class object cannot be cast to a derived class reference. The `std::bad_cast` could be caught as type, irrespective of any value it might represent.

Actually, it is not even necessary for a type to contain values. It is possible to define an *empty enum*, an `enum` without any values, whose name may thereupon be used as a legitimate type name in, e.g. a `catch` clause defining an exception handler.

An empty `enum` is defined as follows (often, but not necessarily within a `class`):

```
enum EmptyEnum
{};
```

Now an `EmptyEnum` may be thrown (and caught) as an exception:

```
#include <iostream>

enum EmptyEnum
{};

using namespace std;

int main()
try
{
    throw EmptyEnum();
}
catch (EmptyEnum)
{
    cout << "Caught empty enum\n";
}
/*
    Generated output:

    Caught empty enum
*/
```

17.5 Revisiting virtual constructors

In section 14.11 the notion of virtual constructors was introduced. In that section a class `Base` was used as an abstract base class. A class `Clonable` was thereupon defined to manage `Base` class pointers in containers like vectors.

As the class Base is a very small class, hardly requiring any implementation, it can well be defined as a nested class in Clonable. This will emphasize the close relationship that exists between Clonable and Base, as shown by the way classes are derived from Base. One no longer writes:

```
class Derived: public Base
```

but rather:

```
class Derived: public Clonable::Base
```

Other than defining Base as a nested class, and deriving from Clonable::Base rather than from Base, nothing needs to be modified. Here is the program shown earlier in section 14.11, but now using nested classes:

```
#include <iostream>
#include <vector>
#include <typeinfo>

class Clonable
{
    public:
        class Base
        {
            public:
                virtual ~Base();
                virtual Base *clone() const = 0;
        };

    private:
        Base *d_bp;

    public:
        Clonable();
        ~Clonable();
        Clonable(Clonable const &other);
        Clonable &operator=(Clonable const &other);

        // New for virtual constructions:
        Clonable(Base const &bp);
        Base &get() const;

    private:
        void copy(Clonable const &other);
};


inline Clonable::Base::~Base()
{}

inline Clonable::Clonable()
:
    d_bp(0)
{}
inline Clonable::~Clonable()
{
    delete d_bp;
}
inline Clonable::Clonable(Clonable const &other)
```

```
{
    copy(other);
}
inline Clonable &Clonable::operator=(Clonable const &other)
{
    if (this != &other)
    {
        delete d_bp;
        copy(other);
    }
    return *this;
}

inline Clonable::Clonable(Base const &bp)
{
    d_bp = bp.clone();      // allows initialization from
}                           // Base and derived objects

inline Clonable::Base &Clonable::get() const
{
    return *d_bp;
}

inline void Clonable::copy(Clonable const &other)
{
    if ((d_bp = other.d_bp))
        d_bp = d_bp->clone();
}

class Derived1: public Clonable::Base
{
    public:
        ~Derived1();
        virtual Clonable::Base *clone() const;
};

inline Derived1::~Derived1()
{
    std::cout << "~Derived1() called\n";
}
inline Clonable::Base *Derived1::clone() const
{
    return new Derived1(*this);
}

using namespace std;

int main()
{
    vector<Clonable> bv;

    bv.push_back(Derived1());
    cout << "==\n";

    cout << typeid(bv[0].get()).name() << endl;
    cout << "==\n";

    vector<Clonable> v2(bv);
    cout << typeid(v2[0].get()).name() << endl;
```

```
        cout << "==\n";
    }
```

Chapter 18

The Standard Template Library

The `Standard Template Library` (STL) is a general purpose library consisting of containers, generic algorithms, iterators, function objects, allocators, adaptors and data structures. The data structures used in the algorithms are *abstract* in the sense that the algorithms can be used on (practically) every data type.

The algorithms can work on these abstract data types due to the fact that they are *template* based algorithms. In this chapter the *construction* of templates is not discussed (see chapter 20 for that). Rather, this chapter focuses on the *use* of these template algorithms.

Several parts of the standard template library have already been discussed in the **C++** Annotations. In chapter 12 the abstract containers were discussed, and in section 10.10 function objects were introduced. Also, *iterators* were mentioned at several places in this document.

The remaining components of the STL will be covered in this and the next chapter. Iterators, adaptors, smart pointers, multi threading and other features of the STL will be discussed in the coming sections. Generic algorithms are covered in the next chapter (19).

Allocators take care of the memory allocation within the STL. The default allocator class suffices for most applications, and is not further discussed in the **C++** Annotations.

All elements of the STL are defined in the standard namespace. Therefore, a `using namespace std` or comparable directive is required unless it is preferred to specify the required namespace explicitly. This occurs in at least one situation: in header files no `using` directive should be used, so here the `std::` scope specification should always be specified when referring to elements of the STL.

18.1 Predefined function objects

Function objects play important roles in combination with generic algorithms. For example, there exists a generic algorithm `sort()` expecting two iterators defining the range of objects that should be sorted, as well as a function object calling the appropriate comparison operator for two objects. Let's take a quick look at this situation. Assume strings are stored in a vector, and we want to sort the vector in descending order. In that case, sorting the vector `stringVec` is as simple as:

```
sort(stringVec.begin(), stringVec.end(), greater<std::string>());
```

The last argument is recognized as a *constructor*: it is an *instantiation* of the `greater<>()` class template, applied to `strings`. This object is called as a function object by the `sort()` generic algorithm. It will call the `operator>()` of the provided data type (here `std::string`) whenever its `operator()()` is called. Eventually, when `sort()` returns, the first element of the vector will be the greatest element.

The `operator()()` (function call operator) itself is *not* visible at this point: don't confuse the parentheses in `greater<string>()` with calling `operator()()`. When that operator is actually used inside `sort()`, it receives two arguments: two strings to compare for 'greaterness'. Internally, the `operator>()` of the data type to which the iterators point (i.e., `string`) is called by `greater<string>`'s function operator (`operator()()`) to compare the two objects. Since `greater<>`'s function call operator is defined inline, the call itself is not actually present in the code. Rather, `sort()` calls `string::operator>()`, thinking it called `greater<>::operator()()`.

Now that we know that a constructor is passed as argument to (many) generic algorithms, we can design our own function objects. Assume we want to sort our vector case-insensitively. How do we proceed? First we note that the default `string::operator<()` (for an incremental sort) is not appropriate, as it does case sensitive comparisons. So, we provide our own `case_less` class, in which the two strings are compared case insensitively. Using the standard **C** function `strcasecmp()`, the following program performs the trick. It sorts its command-line arguments in ascending alphabetic order:

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

class case_less
{
    public:
        bool operator()(string const &left, string const &right) const
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

int main(int argc, char **argv)
{
    sort(argv, argv + argc, case_less());
    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;
}
```

The default constructor of the `class case_less` is used with `sort()`'s final argument. Therefore, the only member function that must be defined with the class `case_less` is the function object operator `operator()()`. Since we know it's called with `string` arguments, we define it to expect two `string` arguments, which are used in the `strcasecmp()` function. Furthermore, the `operator()()` function is made inline, so that it does not produce overhead when called by the `sort()` function. The `sort()` function calls the function object with various combinations of `strings`, i.e., it *thinks* it does so. However, in fact it calls `strcasecmp()`, due to the inline-nature of `case_less::operator()()`.

The comparison function object is often a *predefined function object*, since these are available for many commonly used operations. In the following sections the available predefined function objects are presented, together with some examples showing their use. At the end of the section about function objects *function adaptors* are introduced. Before predefined function objects can be used the following preprocessor directive must have been specified:

```
#include <functional>
```

Predefined function objects are used predominantly with generic algorithms. Predefined function objects exists for arithmetic, relational, and logical operations. In section 23.5 predefined function objects are developed performing bitwise operations.

18.1.1 Arithmetic function objects

The arithmetic function objects support the standard arithmetic operations: addition, subtraction, multiplication, division, modulus and negation. These predefined arithmetic function objects invoke the corresponding operator of the associated data type. For example, for addition the function object `plus<Type>` is available. If we set `type` to `size_t` then the + operator for `size_t` values is used, if we set `type` to `string`, then the + operator for strings is used. For example:

```
#include <iostream>
#include <string>
#include <functional>
using namespace std;

int main(int argc, char **argv)
{
    plus<size_t> uAdd;       // function object to add size_ts

    cout << "3 + 5 = " << uAdd(3, 5) << endl;

    plus<string> sAdd;       // function object to add strings

    cout << "argv[0] + argv[1] = " << sAdd(argv[0], argv[1]) << endl;
}
/*
    Generated output with call: a.out going

    3 + 5 = 8
    argv[0] + argv[1] = a.outgoing
*/
```

Why is this useful? Note that the function object can be used with all kinds of data types (not only with the predefined datatypes), in which the particular operator has been overloaded. Assume that we want to perform an operation on a common variable on the one hand and, on the other hand, in turn on each element of an array. E.g., we want to compute the sum of the elements of an array; or we want to concatenate all the strings in a text-array. In situations like these the function objects come in handy. As noted before, the function objects are heavily used in the context of the generic algorithms, so let's take a quick look ahead at one of them.

One of the generic algorithms is called `accumulate()`. It visits all elements implied by an iterator-range, and performs a requested binary operation on a common element and each of the elements in the range, returning the accumulated result after visiting all elements. For example, the following program accumulates all command line arguments, and prints the final string:

```
#include <iostream>
#include <string>
#include <functional>
#include <numeric>
using namespace std;

int main(int argc, char **argv)
{
    string result =
            accumulate(argv, argv + argc, string(), plus<string>());

    cout << "All concatenated arguments: " << result << endl;
}
```

The first two arguments define the (iterator) range of elements to visit, the third argument is `string()`. This anonymous string object provides an initial value. It could as well have been initialized to

```
        string("All concatenated arguments: ")
```

in which case the `cout` statement could have been a simple

```
    cout << result << endl;
```

Then, the operator to apply is `plus<string>()`. Note here that a constructor is called: it is *not* `plus<string>`, but rather `plus<string>()`. The final concatenated string is returned.

Now we define our own class `Time`, in which the `operator+()` has been overloaded. Again, we can apply the predefined function object `plus`, now tailored to our newly defined datatype, to add times:

```
    #include <iostream>
    #include <sstream>
    #include <string>
    #include <vector>
    #include <functional>
    #include <numeric>

    using namespace std;

    class Time
    {
        friend ostream &operator<<(ostream &str, Time const &time)
        {
            return cout << time.d_days << " days, " << time.d_hours <<
                                                       " hours, " <<
                           time.d_minutes << " minutes and " <<
                           time.d_seconds << " seconds.";
        }

        size_t d_days;
        size_t d_hours;
        size_t d_minutes;
        size_t d_seconds;

        public:
            Time(size_t hours, size_t minutes, size_t seconds)
            :
                d_days(0),
                d_hours(hours),
                d_minutes(minutes),
                d_seconds(seconds)
            {}
            Time &operator+=(Time const &rValue)
            {
                d_seconds   += rValue.d_seconds;
                d_minutes   += rValue.d_minutes   + d_seconds / 60;
                d_hours     += rValue.d_hours     + d_minutes / 60;
                d_days      += rValue.d_days      + d_hours   / 24;
                d_seconds   %= 60;
                d_minutes   %= 60;
                d_hours     %= 24;

                return *this;
            }
    };
    Time const operator+(Time const &lValue, Time const &rValue)
```

```
{
    return Time(lValue) += rValue;
}

int main(int argc, char **argv)
{
    vector<Time> tvector;

    tvector.push_back(Time( 1, 10, 20));
    tvector.push_back(Time(10, 30, 40));
    tvector.push_back(Time(20, 50,  0));
    tvector.push_back(Time(30, 20, 30));

    cout <<
        accumulate
        (
            tvector.begin(), tvector.end(), Time(0, 0, 0), plus<Time>()
        ) <<
        endl;
}
/*
    produced output:

    2 days, 14 hours, 51 minutes and 30 seconds.
*/
```

Note that all member functions of `Time` in the above source are inline functions. This approach was followed in order to keep the example relatively small and to show explicitly that the `operator+=()` function may be an inline function. On the other hand, in real life `Time`'s `operator+=()` should probably not be made inline, due to its size.

Considering the previous discussion of the `plus` function object, the example is pretty straightforward. The class `Time` defines a constructor, it defines an insertion operator and it defines its own `operator+()`, adding two time objects.

In `main()` four `Time` objects are stored in a `vector<Time>` object. Then, the `accumulate()` generic algorithm is called to compute the accumulated time. It returns a `Time` object, which is inserted in the `cout ostream` object.

While the first example did show the use of a *named* function object, the last two examples showed the use of *anonymous* objects which were passed to the (`accumulate()`) function.

The following arithmetic objects are available as predefined objects:

- `plus<>()`: as shown, this object's `operator()()` member calls `operator+()` as a binary operator, passing it its two parameters, returning `operator+()`'s return value.
- `minus<>()`: this object's `operator()()` member calls `operator-()` as a binary operator, passing it its two parameters and returning `operator-()`'s return value.
- `multiplies<>()`: this object's `operator()()` member calls `operator*()` as a binary operator, passing it its two parameters and returning `operator*()`'s return value.
- `divides<>()`: this object's `operator()()` member calls `operator/()`, passing it its two parameters and returning `operator/()`'s return value.
- `modulus<>()`: this object's `operator()()` member calls `operator%()`, passing it its two parameters and returning `operator%()`'s return value.
- `negate<>()`: this object's `operator()()` member calls `operator-()` as a unary operator, passing it its parameter and returning the unary `operator-()`'s return value.

An example using the unary `operator-()` follows, in which the `transform()` generic algorithm is used to toggle the signs of all elements in an array. The `transform()` generic algorithm expects two iterators, defining the range of objects to be transformed, an iterator defining the begin of the destination range (which may be the same iterator as the first argument) and a function object defining a unary operation for the indicated data type.

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
    int iArr[] = { 1, -2, 3, -4, 5, -6 };

    transform(iArr, iArr + 6, iArr, negate<int>());

    for (int idx = 0; idx < 6; ++idx)
        cout << iArr[idx] << ", ";

    cout << endl;
}
/*
    Generated output:

    -1, 2, -3, 4, -5, 6,
*/
```

18.1.2 Relational function objects

The relational operators are called by the relational function objects. All standard relational operators are supported: `==`, `!=`, `>`, `>=`, `<` and `<=`. The following objects are available:

- `equal_to<>()`: this object's `operator()()` member calls `operator==()` as a binary operator, passing it its two parameters and returning `operator==()`'s return value.
- `not_equal_to<>()`: this object's `operator()()` member calls `operator!=()` as a binary operator, passing it its two parameters and returning `operator!=()`'s return value.
- `greater<>()`: this object's `operator()()` member calls `operator>()` as a binary operator, passing it its two parameters and returning `operator>()`'s return value.
- `greater_equal<>()`: this object's `operator()()` member calls `operator>=()` as a binary operator, passing it its two parameters and returning `operator>=()`'s return value.
- `less<>()`: this object's `operator()()` member calls `operator<()` as a binary operator, passing it its two parameters and returning `operator<()`'s return value.
- `less_equal<>()`: this object's `operator()()` member calls `operator<=()` as a binary operator, passing it its two parameters and returning `operator<=()`'s return value.

Like the arithmetic function objects, these function objects can be used as *named* or as *anonymous* objects. An example using the relational function objects using the generic algorithm `sort()` is:

```
#include <iostream>
#include <string>
#include <functional>
```

```
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
    sort(argv, argv + argc, greater_equal<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;

    sort(argv, argv + argc, less<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;
}
```

The `sort()` generic algorithm expects an iterator range and a comparator of the data type to which the iterators point. The example shows the alphabetic sorting of strings and the reversed sorting of strings. By passing `greater_equal<string>()` the strings are sorted in *decreasing* order (the first word will be the 'greatest'), by passing `less<string>()` the strings are sorted in *increasing* order (the first word will be the 'smallest').

Note that the type of the elements of `argv` is `char *`, and that the relational function object expects a `string`. The relational object `greater_equal<string>()` will therefore use the `>=` operator of strings, but will be called with `char *` variables. The promotion from `char const *` to `string` is performed silently.

18.1.3 Logical function objects

The logical operators are called by the logical function objects. The standard logical operators are supported: `and`, `or`, and `not`. The following objects are available:

- `logical_and<>()`: this object's `operator()()` member calls `operator&&()` as a binary operator, passing it its two parameters and returning `operator&&()`'s return value.
- `logical_or<>()`: this object's `operator()()` member calls `operator||()` as a binary operator, passing it its two parameters and returning `operator||()`'s return value.
- `logical_not<>()`: this object's `operator()()` member calls `operator!()` as a unary operator, passing it its parameter and returning the unary `operator!()`'s return value.

An example using `operator!()` is provided in the following trivial program, in which the `transform()` generic algorithm is used to transform the logicalvalues stored in an array:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
    bool bArr[] = {true, true, true, false, false, false};
    size_t const bArrSize = sizeof(bArr) / sizeof(bool);
```

```
    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;

    transform(bArr, bArr + bArrSize, bArr, logical_not<bool>());

    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;
}
/*
    generated output:

    1 1 1 0 0 0
    0 0 0 1 1 1
*/
```

18.1.4 Function adaptors

Function adaptors modify the working of existing function objects. There are two kinds of function adaptors:

- *Binders* are function adaptors converting binary function objects to unary function objects. They do so by *binding* one object to a constant function object. For example, with the `minus<int>()` function object, which is a binary function object, the first argument may be bound to 100, meaning that the resulting value will always be `100` minus the value of the second argument. Either the first or the second argument may be bound to a specific value. To bind the first argument to a specific value, the function object `bind1st()` is used. To bind the second argument of a binary function to a specific value `bind2nd()` is used. As an example, assume we want to count all elements of a vector of `Person` objects that exceed (according to some criterion) some reference `Person` object. For this situation we pass the following binder and relational function object to the `count_if()` generic algorithm:

  ```
  bind2nd(greater<Person>(), referencePerson)
  ```

  What would such a binder do? First of all, it's a function object, so it needs `operator()()`. Next, it expects two arguments: a reference to another function object and a fixed operand. Although binders are defined as templates, it is illustrative to have a look at their implementations, assuming they were straight functions. Here is such a pseudo-implementation of a binder:

  ```
  class bind2nd
  {
      FunctionObject const &d_object;
      Operand const &d_operand;
      public:
          bind2nd(FunctionObject const &object, Operand const &operand);
          ReturnType operator()(Operand const &lvalue);
  };
  inline bind2nd::bind2nd(FunctionObject const &object,
                          Operand const &operand)
  :
      d_object(object),
      d_operand(operand)
  {}
  inline ReturnType bind2nd::operator()(Operand const &lvalue)
  {
      return d_object(lvalue, d_rvalue);
  }
  ```

When its `operator()()` member is called the binder merely passes the call to the object's `operator()()`, providing it with two arguments: the `lvalue` it itself received and the fixed operand it received via its constructor. Note the simplicity of these kind of classes: all its members can usually be implemented inline.

The `count_if()` generic algorithm visits all the elements in an iterator range, returning the number of times the predicate specified as its final argument returns `true`. Each of the elements of the iterator range is given to the predicate, which is therefore a unary function. By using the binder the binary function object `greater()` is adapted to a unary function object, comparing each of the elements in the range to the reference person. Here is, to be complete, the call of the `count_if()` function:

```
count_if(pVector.begin(), pVector.end(),
    bind2nd(greater<Person>(), referencePerson))
```

- *Negators* are function adaptors converting the truth value of a predicate function. Since there are unary and binary predicate functions, there are two negator function adaptors: `not1()` is the negator used with unary function objects, `not2()` is the negator used with binary function objects.

If we want to count the number of persons in a `vector<Person>` vector *not* exceeding a certain reference person, we may, among other approaches, use either of the following alternatives:

- Use a binary predicate that directly offers the required comparison:

```
count_if(pVector.begin(), pVector.end(),
    bind2nd(less_equal<Person>(), referencePerson))
```

- Use `not2` combined with the `greater()` predicate:

```
count_if(pVector.begin(), pVector.end(),
    bind2nd(not2(greater<Person>()), referencePerson))
```

Note that `not2()` is a negator negating the truth value of a binary `operator()()` member: it must be used to wrap the binary predicate `greater<Person>()`, negating its truth value.

- Use `not1()` combined with the `bind2nd()` predicate:

```
count_if(pVector.begin(), pVector.end(),
    not1(bind2nd(greater<Person>(), referencePerson)))
```

Note that `not1()` is a negator negating the truth value of a unary `operator()()` member: it is used to wrap the unary predicate `bind2nd()`, negating its truth value.

The following little example illustrates the use of negator function adaptors, completing the section on function objects:

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;

int main(int argc, char **argv)
{
    int iArr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    cout << count_if(iArr, iArr + 10, bind2nd(less_equal<int>(), 6)) <<
        endl;
    cout << count_if(iArr, iArr + 10, bind2nd(not2(greater<int>()), 6)) <<
        endl;
```

```
    cout << count_if(iArr, iArr + 10, not1(bind2nd(greater<int>(), 6))) <<
        endl;
}
/*
    produced output:

    6
    6
    6
*/
```

One may wonder which of these alternative approaches is fastest. Using the first approach, in which a directly available function object was used, two actions must be performed for each iteration by `count_if()`:

- The binder's `operator()()` is called;
- The operation <= is performed for `int` values.

Using the second approach, in which the `not2` negator is used to negate the truth value of the complementary logicalfunction adaptor, three actions must be performed for each iteration by `count_if()`:

- The binder's `operator()()` is called;
- The negator's `operator()()` is called;
- The operation > is performed for `int` values.

Using the third approach, in which a `not1` negator is used to negate the truth value of the binder, three actions must be performed for each iteration by `count_if()`:

- The negator's `operator()()` is called;
- The binder's `operator()()` is called;
- The operation > is performed for `int` values.

From this, one might deduce that the first approach is fastest. Indeed, using Gnu's `g++` compiler on an old, 166 MHz pentium, performing 3,000,000 `count_if()` calls for each variant, shows the first approach requiring about 70% of the time needed by the other two approaches to complete.

However, these differences disappear if the compiler is instructed to optimize for speed (using the `-O6` compiler flag). When interpreting these results one should keep in mind that multiple nested function calls are merged into a single function call if the implementations of these functions are given inline and if the compiler follows the suggestion to implement these functions as true inline functions indeed. If this is happening, the three approaches all merge to a single operation: the comparison between two `int` values. It is likely that the compiler does so when asked to optimize for speed.

18.2 Iterators

Iterators are objects acting like pointers. Iterators have the following general characteristics:

- Two iterators may be compared for (in)equality using the == and != operators. Note that the *ordering* operators (e.g., >, <) normally cannot be used.
- Given an iterator `iter`, `*iter` represents the object the iterator points to (alternatively, `iter->` can be used to reach the members of the object the iterator points to).

- `++iter` or `iter++` advances the iterator to the next element. The notion of advancing an iterator to the next element is consequently applied: several containers have a *reversed_iterator* type, in which the `iter++` operation actually reaches a previous element in a sequence.
- *Pointer arithmetic* may be used with containers having their elements stored consecutively in memory. This includes the `vector` and `deque`. For these containers `iter + 2` points to the second element beyond the one to which `iter` points.
- An interator which is merely defined is comparable to a 0-pointer, as shown by the following little example:

```
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector<int>::iterator vi;

    cout << &*vi << endl;       // prints 0
}
```

The STL containers usually define members producing iterators (i.e., type `iterator`) using member functions `begin()` and `end()` and, in the case of reversed iterators (type `reverse_iterator`), `rbegin()` and `rend()`. Standard practice requires the iterator range to be *left inclusive*: the notation `[left, right)` indicates that `left` is an iterator pointing to the first element that is to be considered, while `right` is an iterator pointing just *beyond* the last element to be used. The iterator-range is said to be *empty* when `left == right`. Note that with empty containers the begin- and end-iterators are equal to each other.

The following example shows a situation where all elements of a vector of strings are written to `cout` using the iterator range `[begin(), end())`, and the iterator range `[rbegin(), rend())`. The following example shows that the `for-loops` for both ranges are identical. Furthermore it nicely illustrates how the `auto` keyword can be used to define the type of the loop control variable instead of using a much more verbose explicit variable definition like `vector<string>::iterator` (see also section 3.3.5):

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char **argv)
{
    vector<string> args(argv, argv + argc);

    for (auto iter = args.begin(); iter != args.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    for (auto iter = args.rbegin(); iter != args.rend(); ++iter)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

Furthermore, the STL defines *const_iterator* types to be able to visit a series of elements in a constant container. Whereas the elements of the vector in the previous example could have been altered, the elements of the vector in the next example are immutable, and `const_iterators` are required:

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char **argv)
{
    vector<string> const args(argv, argv + argc);

    for
    (
        vector<string>::const_iterator iter = args.begin();
            iter != args.end();
                ++iter
    )
        cout << *iter << " ";
    cout << endl;

    for
    (
        vector<string>::const_reverse_iterator iter = args.rbegin();
            iter != args.rend();
                ++iter
    )
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

The examples also illustrates that plain pointers can be used instead of iterators. The initialization `vector<string> args(argv, argv + argc)` provides the `args` vector with a pair of pointer-based iterators: `argv` points to the first element to initialize `sarg` with, `argv + argc` points just beyond the last element to be used, `argv++` reaches the next string. This is a general characteristic of pointers, which is why they too can be used in situations where `iterators` are expected.

The STL defines five types of iterators. These types recur in the generic algorithms, and in order to be able to create a particular type of iterator yourself it is important to know their characteristics. In general, iterators must define:

- `operator==()`, testing two iterators for equality,
- `operator++()`, incrementing the iterator, as prefix operator,
- `operator*()`, to access the element the iterator refers to,

The following types of iterators are used when describing generic algorithms later in this chapter:

- **InputIterators**.

  InputIterators can read from a container. The dereference operator is guaranteed to work as `rvalue` in expressions. Instead of an InputIterator it is also possible to (see below) use a Forward-, Bidirectional- or RandomAccessIterator. With the generic algorithms presented in this chapter. Notations like `InputIterator1` and `InputIterator2` may be observed as well. In these cases, numbers are used to indicate which iterators 'belong together'. E.g., the generic function `inner_product()` has the following prototype:

  ```
  Type inner_product(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, Type init);
  ```

Here `InputIterator1 first1` and `InputIterator1 last1` are a set of input iterators defining one range, while `InputIterator2 first2` defines the beginning of a second range. Analogous notations like these may be observed with other iterator types.

- **OutputIterators**:

  OutputIterators can be used to write to a container. The dereference operator is guaranteed to work as an `lvalue` in expressions, but not necessarily as `rvalue`. Instead of an OutputIterator it is also possible to use, see below, a Forward-, Bidirectional- or RandomAccessIterator.

- **ForwardIterators**:

  ForwardIterators combine InputIterators and OutputIterators. They can be used to traverse containers in one direction, for reading and/or writing. Instead of a ForwardIterator it is also possible to use a Bidirectional- or RandomAccessIterator.

- **BidirectionalIterators**:

  BidirectionalIterators can be used to traverse containers in both directions, for reading and writing. Instead of a BidirectionalIterator it is also possible to use a RandomAccessIterator. For example, to traverse a list or a deque a BidirectionalIterator may be useful.

- **RandomAccessIterators**:

  RandomAccessIterators provide random access to container elements. An algorithm such as `sort()` requires a RandomAccessIterator, and can therefore *not* be used with lists or maps, which only provide BidirectionalIterators.

The example given with the RandomAccessIterator illustrates how to approach iterators: look for the iterator that's required by the (generic) algorithm, and then see whether the datastructure supports the required iterator. If not, the algorithm cannot be used with the particular datastructure.

18.2.1 Insert iterators

Generic algorithms often require a target container into which the results of the algorithm are deposited. For example, the `copy()` algorithm has three parameters, the first two defining the range of visited elements, and the third parameter defines the first position where the results of the copy operation should be stored. With the `copy()` algorithm the number of elements that are copied are usually available beforehand, since the number is usually determined using pointer arithmetic. However, there are situations where pointer arithmetic cannot be used. Analogously, the number of resulting elements sometimes differs from the number of elements in the initial range. The generic algorithm `unique_copy()` is a case in point: the number of elements which are copied to the destination container is normally not known beforehand.

In situations like these, an `inserter` adaptor function may be used to create elements in the destination container when they are needed. There are three types of `inserter()` adaptors:

- `back_inserter()`: calls the container's `push_back()` member to add new elements at the end of the container. E.g., to copy all elements of `source` in reversed order to the back of `destination`:

  ```
  copy(source.rbegin(), source.rend(), back_inserter(destination));
  ```

- `front_inserter()` calls the container's `push_front()` member to add new elements at the beginning of the container. E.g., to copy all elements of `source` to the front of the destination container (thereby also reversing the order of the elements):

  ```
  copy(source.begin(), source.end(), front_inserter(destination));
  ```

- `inserter()` calls the container's `insert()` member to add new elements starting at a specified starting point. E.g., to copy all elements of `source` to the destination container, starting at the beginning of `destination`, shifting existing elements beyond the newly inserted elements:

```
copy(source.begin(), source.end(), inserter(destination,
    destination.begin()));
```

Concentrating on the `back_inserter()`, this iterator expects the name of a container having a member `push_back()`. This member is called by the inserter's `operator()()` member. When a class (other than the abstract containers) supports a `push_back()` container, its objects can also be used as arguments of the `back_inserter()` if the class defines a

```
typedef DataType const &const_reference;
```

in its interface, where `DataType const &` is the type of the parameter of the class's member function `push_back()`. For example, the following program defines a (compilable) skeleton of a class `IntStore`, whose objects can be used as arguments of the `back_inserter` iterator:

```
#include <algorithm>
#include <iterator>
using namespace std;

class Y
{
    public:
        typedef int const &const_reference;

        void push_back(int const &)
        {}
};

int main()
{
    int arr[] = {1};
    Y y;

    copy(arr, arr + 1, back_inserter(y));
}
```

18.2.2 Iterators for 'istream' objects

The `istream_iterator<Type>()` can be used to define an iterator (pair) for `istream` objects. The general form of the `istream_iterator<Type>()` iterator is:

```
istream_iterator<Type> identifier(istream &inStream)
```

Here, `Type` is the type of the data elements that are read from the `istream` stream. `Type` may be any type for which `operator>>` is defined with `istream` objects.

The default constructor defines the end of the iterator pair, corresponding to end-of-stream. For example,

```
istream_iterator<string> endOfStream;
```

Note that the actual *stream* object which was specified for the begin-iterator is *not* mentioned here.

Using a back_inserter() and a set of istream_iterator<>() adaptors, all strings could be read from cin as follows:

```
#include <algorithm>
#include <iterator>
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string> vs;

    copy(istream_iterator<string>(cin), istream_iterator<string>(),
         back_inserter(vs));

    for
    (
        vector<string>::iterator from = vs.begin();
            from != vs.end();
                ++from
    )
        cout << *from << " ";
    cout << endl;

    return 0;
}
```

In the above example, note the use of the anonymous versions of the istream_iterator adaptors. Especially note the use of the anonymous default constructor. The following (non-anonymous) construction could have been used instead of istream_iterator<string>():

```
istream_iterator<string> eos;

copy(istream_iterator<string>(cin), eos, back_inserter(vs));
```

Before istream_iterators can be used the following preprocessor directive must have been specified:

```
#include <iterator>
```

This is implied when iostream is included.

18.2.3 Iterators for 'istreambuf' objects

Input iterators are also available for streambuf objects. Before istreambuf_iterators can be used the following preprocessor directive must have been specified:

```
#include <iterator>
```

The istreambuf_iterator is available for reading from streambuf objects supporting input operations. The standard operations that are available for istream_iterator objects are also available for istreambuf_iterators. There are three constructors:

- `istreambuf_iterator<Type>()`:

  This constructor represents the end-of-stream iterator while extracting values of type `Type` from the `streambuf`.

- `istreambuf_iterator<Type>(istream)`:

  This constructor constructs an `istreambuf_iterator` accessing the `streambuf` of the `istream` object, used as the constructor's argument.

- `istreambuf_iterator<Type>(streambuf *)`:

  This constructor constructs an `istreambuf_iterator` accessing the `streambuf` whose address is used as the constructor's argument.

In section 18.2.4.1 an example is given using both `istreambuf_iterators` and `ostreambuf_iterators`.

18.2.4 Iterators for 'ostream' objects

The `ostream_iterator<Type>()` can be used to define a destination iterator for an `ostream` object. The general forms of the `ostream_iterator<Type>()` iterator are:

```
ostream_iterator<Type> identifier(ostream &outStream), // and:
ostream_iterator<Type> identifier(ostream &outStream, char const *delim);
```

`Type` is the type of the data elements that should be written to the `ostream` stream. `Type` may be any type for which `operator<<` is defined in combinations with `ostream` objects. The latter form of the `ostream_iterators` separates the individual `Type` data elements by `delimiter` strings. The former definition does not use any delimiters.

The following example shows how `istream_iterators` and an `ostream_iterator` may be used to copy information of a file to another file. A subtlety is the statement `in.unsetf(ios::skipws)`: it resets the `ios::skipws` flag. The consequence of this is that the default behavior of `operator>>`, to skip whitespace, is modified. White space characters are simply returned by the operator, and the file is copied unrestrictedly. Here is the program:

Before `ostream_iterators` can be used the following preprocessor directive must have been specified:

```
#include <iterator>
```

18.2.4.1 Iterators for 'ostreambuf' objects

Before an `ostreambuf_iterator` can be used the following preprocessor directive must have been specified:

```
#include <iterator>
```

The `ostreambuf_iterator` is available for writing to `streambuf` objects supporting output operations. The standard operations that are available for `ostream_iterator` objects are also available for `ostreambuf_iterators`. There are two constructors:

- `ostreambuf_iterator<Type>(ostream)`:

  This constructor constructs an `ostreambuf_iterator` accessing the `streambuf` of the `ostream` object, used as the constructor's argument, to insert values of type `Type`.

- `ostreambuf_iterator<Type>(streambuf *)`:

  This constructor constructs an `ostreambuf_iterator` accessing the `streambuf` whose address is used as the constructor's argument.

Here is an example using both `istreambuf_iterators` and an `ostreambuf_iterator`, showing yet another way to copy a stream:

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    istreambuf_iterator<char> in(cin.rdbuf());
    istreambuf_iterator<char> eof;
    ostreambuf_iterator<char> out(cout.rdbuf());

    copy(in, eof, out);

    return 0;
}
```

18.3 The class 'unique_ptr' (C++0x)

One of the problems using pointers is that strict bookkeeping is required with respect to their memory use and lifetime. When a pointer variable goes out of scope, the memory pointed to by the pointer is suddenly inaccessible, and the program suffers from a memory leak. For example, in the following function `fun()`, a memory leak is created by calling `fun()`: the allocated `int` value remains inaccessibly allocated:

```
void fun()
{
    new int;
}
```

To prevent memory leaks strict bookkeeping is required: the programmer has to make sure that the memory pointed to by a pointer is deleted just before the pointer variable goes out of scope. The above example could easily be repaired:

```
void fun()
{
    delete new int;
}
```

Now `fun()` merely wastes some time.

When a pointer variable points to *a single value or object*, the bookkeeping requirements may be relaxed when the pointer variable is defined as a `std::unique_ptr` object. Unique_ptrs are *objects*, masquerading as pointers. Since they're objects, their destructors are called when they go out of scope, and because of that, their destructors will take the responsibility of deleting the dynamically allocated memory.

Before `unique_ptrs` can be used the following preprocessor directive must have been specified:

```
#include <memory>
```

Normally, an unique_ptr object is initialized with a dynamically created value or object.

The following should be considered when using unique_ptrs:

- when assigning a unique_ptr to another *move semantics* is used. If move semantics are not available compilation will fail. On the other hand, if compilation succeeds then the used containers or generic algorithms support the use of unique_ptrs. Here is an example:

```
std::unique_ptr<int> up1(new int);
std::unique_ptr<int> up2 = up1;      // compilation error
```

  The second definition fails to compile since unique_ptr's copy constructor is private (the same holds true for the assignment operator). The unique_ptr class *does* offer facilities to initialize and assign from *rvalue references*:

```
class unique_ptr                        // interface partially shown
{
    public:
        unique_ptr(unique_ptr &&other); // rvalues bind here
    private:
        unique_ptr(const unique_ptr &other);
};
```

  Consequently, in the above example move semantics should be used. E.g.,

```
std::unique_ptr<int> up1(new int);
std::unique_ptr<int> up2 = std::move(up3);
```

- a unique_ptr object should only point to memory that was made available dynamically, as only dynamically allocated memory can be deleted.
- multiple unique_ptr objects should not be allowed to point to the same block of dynamically allocated memory. The unique_ptr's interface was designed to prevent this from happening. Once an unique_ptr object goes out of scope, it deletes the memory it points to, immediately changing any other object also pointing to the allocated memory into a wild pointer.

The class unique_ptr defines several member functions to access the pointer itself or to have the unique_ptr point to another block of memory. These member functions and ways to construct unique_ptr objects are discussed in the next sections.

A unique_ptr (as well as a shared_ptr, see section 18.4) can be used as a safe alternative to the now deprecated auto_ptr. It also augments auto_ptr as it can be used with containers and algorithms; as it adds customizable deleters and as it adds the possibility to handle arrays.

18.3.1 Defining 'unique_ptr' objects (C++0x)

There are three ways to define unique_ptr objects. Each definition contains the usual <type> specifier between angle brackets. Concrete examples are given in the coming sections, but an overview of the various possibilities is presented here:

- The basic form initializes an unique_ptr object to point to a block of memory allocated by the new operator:

```
unique_ptr<type> identifier (new-expression [, deleter]);
```

  This form is discussed in section 18.3.2.

- Another form initializes an `unique_ptr` object using *move semantics*, either supported by the data type itself or forced using `std::move`:

```
// type must support move semantics or std::move(other) must be used
unique_ptr<type> identifier(another unique_ptr for type);
```

  This form is discussed in section 18.3.4.

- The third form simply creates an `unique_ptr` object that does not point to a particular block of memory:

```
unique_ptr<type> identifier;
```

  This form is discussed in section 18.3.5.

18.3.2 Pointing to a newly allocated object (C++0x)

The basic form to initialize an `unique_ptr` object is to provide its constructor with a block of memory allocated by `operator new` operator. The generic form is:

```
unique_ptr<type [, deleter_type]> identifier(new-expression
        [, deleter = deleter_type()]);
```

The second (template) argument (`deleter(_type)`) is optional and may refer to a class/object handling the destruction of the allocated memory. This is used, e.g., in situations where a double pointer is allocated and the destruction must visit each nested pointer to destroy the memory it points at (see below).

Here is an example initializing an `unique_ptr` pointing to a `string` object:

```
unique_ptr<string> strPtr(new string("Hello world"));
```

To initialize an `unique_ptr` to point to a `double` value the following construction can be used:

```
unique_ptr<double> dPtr(new double(123.456));
```

Note the use of `operator new` in the above expressions. Using `new` ensures the dynamic nature of the memory pointed to by the `unique_ptr` objects and allows the deletion of the memory once `unique_ptr` objects go out of scope. Also note that `type` does *not* mention the pointer: the type used in the `unique_ptr` construction is the same as used in `new` expressions.

The next example shows how an explicitly defined deleter may be used to delete a dynamically allocated array of pointers to strings. Instead of using a template value parameter the deleter's constructor could of course also be given a parameter initializing the deleter with the number of elements to delete:

```
#include <iostream>
#include <memory>
using namespace std;

template <typename Type, size_t size>
struct D2
  {
    void operator()(Type * ptr) const
    {
        for (size_t idx = 0; idx < size; ++idx)
            delete ptr[idx];
        delete ptr;
```

```
    }
};
int main()
{
    unique_ptr<int *, D2<int *, 10>> sp2(new int *[10]);

    D2<int *, 10> &obj = sp2.get_deleter();
}
```

In the example allocating an `int` values given in section 18.3, the memory leak can be avoided using an `unique_ptr` object:

```
#include <memory>
using namespace std;

void fun()
{
    unique_ptr<int> ip(new int);
}
```

All member functions available for objects allocated by the `new` expression can be reached via the `unique_ptr` as if it was a plain pointer to the dynamically allocated object. For example, in the following program the text '`C++`' is inserted behind the word '`hello`':

```
#include <iostream>
#include <memory>
#include <cstring>
using namespace std;

int main()
{
    unique_ptr<string> sp(new string("Hello world"));

    cout << *sp << endl;
    sp->insert(strlen("Hello "), "C++ ");
    cout << *sp << endl;
}
/*
    produced output:

    Hello world
    Hello C++ world
*/
```

18.3.3 Using 'unique_ptr' objects for arrays (C++0x)

When using `unique_ptrs` to stored arrays the dereferencing operator makes little sense. Conversely, when arrays are handled by smart pointers they will benefit from an index operator.

When using smart pointers (`unique_ptr` and `shared_ptr`, see section 18.4) to handle arrays the following additional syntax is available:

- To define an smart pointer for an array, the `[]` notation is used. E.g.,

```
unique_ptr<int[]> intArr(new int[3]);
```

- The index operator is availabe. E.g.,

```
intArr[2] = intArr[0];
it() The default deleter will use tt(delete[]).
```

18.3.4 Moving another 'unique_ptr' (C++0x)

An `unique_ptr` may also be initialized by another `unique_ptr` object for the same type provided `type` supports *move semantics* or move semantics is forced using `std::move`. The generic form is:

```
unique_ptr<type> identifier(other unique_ptr object);
```

For example, to initialize an `unique_ptr<string>`, given the variable `sp` defined in the previous section, the following construction can be used:

```
unique_ptr<string> sp2(move(sp));
```

This move-construction can be used when the data type specified with `unique_ptr` implements its move-constructor.

Analogously, the assignment operator can be used. An `unique_ptr` object may be assigned to another `unique_ptr` object of the same type, again using move-semantics. For example:

```
#include <iostream>
#include <memory>
#include <string>

using namespace std;

int main()
{
    unique_ptr<string> hello1(new string("Hello world"));
    unique_ptr<string> hello2(move(hello1));
    unique_ptr<string> hello3;

    hello3 = move(hello2);
    cout << // *hello1 << endl <<   // would segfault
            // *hello2 << endl <<   // same
            *hello3 << endl;
}
/*
    Produced output:

    Hello world
*/
```

Looking at the above example, we see that

- `hello1` is initialized as described in the previous section.
- Next `hello2` is defined, and it receives its value from `hello1` using a move constructor initialization. This effectively changes `hello1` into a 0-pointer.
- Then `hello3` is defined as a default `unique_ptr<string>`, but it receives its value through a move-assignment from `hello2`, which then becomes a 0-pointer too.

If, in the example program, `hello1` or `hello2` would be inserted into `cout` a *segmentation fault* would be generated. The reason for this will now be clear: it is caused by dereferencing 0-pointers. In the end, only `hello3` actually points to a `string`.

18.3.5 Creating a plain 'unique_ptr' (C++0x)

We've already seen the third form to create an `unique_ptr` object: Without arguments an empty `unique_ptr` object is constructed not pointing to a particular block of memory:

```
unique_ptr<type> identifier;
```

In this case the underlying pointer is set to `0` (zero). Although the `unique_ptr` object itself is not the pointer, its value *can* be compared to `0` to see if it has been initialized. E.g., code like

```
unique_ptr<int> ip;

if (!ip)
    cout << "0-pointer with a unique_ptr object" << endl;
```

will produce the shown text. Alternatively, the member `get()` is available. This member function, as well as the other member functions of the class `unique_ptr` are described in the next section.

18.3.6 Operators and members (C++0x)

The following operators are defined for the class `unique_ptr`:

- `unique_ptr &unique_ptr<Type>operator=(unique_ptr<Type> &other)`:

  This operator will transfer the memory pointed to by the rvalue `unique_ptr` object to the lvalue `unique_ptr` object using *move semantics*. So, the rvalue object *loses* the memory it pointed at, and turns into a 0-pointer.

- `unique_ptr &unique_ptr<Type>operator bool() const`:

  This operator returns `false` if the `unique_ptr` does not point to memory (i.e., its `get()` member, see below, returns 0). Otherwise, `true` is returned.

- `Type &unique_ptr<Type>operator*()`:

  This operator returns a reference to the information stored in the `unique_ptr` object. It acts like a normal pointer dereference operator.

- `Type *unique_ptr<Type>operator->()`:

  This operator returns a pointer to the information stored in the `unique_ptr` object. Through this operator members of a stored object can be selected. For example:

  ```
  unique_ptr<string> sp(new string("hello"));

  cout << sp->c_str() << endl;
  ```

The following member functions are defined for `unique_ptr` objects:

- `Type *unique_ptr<Type>::get()`:

  This member does the same as `operator->()`: it returns a pointer to the information stored in the `unique_ptr` object. This pointer can be inspected: if it's zero the `unique_ptr` object does not point to any memory. This member cannot be used to let the `unique_ptr` object point to (another) block of memory.

- `Deleter &unique_ptr<Type>::get_deleter()`:

  This member returns a reference to the deleter class object used by the `unique_ptr`.

- `Type *unique_ptr<Type>::release()`:

  This member returns a pointer to the information stored in the `unique_ptr` object, which loses the memory it pointed at (and changes into a 0-pointer). The member can be used to transfer the information stored in the `unique_ptr` object to a plain `Type` pointer. It is the responsibility of the programmer to delete the memory returned by this member function.

- `void unique_ptr<Type>::reset(Type *)`:

  This member may also be called *without* argument, to delete the memory stored in the `unique_ptr` object, or with a pointer to a dynamically allocated block of memory, which will thereupon be the memory accessed by the `unique_ptr` object. This member function can be used to assign a new block of memory (new content) to an `unique_ptr` object.

- `void unique_ptr<Type>::swap(unique_ptr<Type> &&)`:

  This member is used to swap two identically typed `unique_ptrs`.

18.3.7 The legacy class 'auto_ptr' (deprecated)

C++ has traditionally offered the (with the advent of the C++ standard deprecated) smart pointer class `std::auto_ptr<Type>`. This class did not support *move semantics* although when one `auto_ptr` object was assigned to another, the right-hand object lost the information it originally pointed at.

The `std::unique_ptr`, available since the C++0x standard does not have `auto_ptr`'s drawbacks and consequently using `auto_ptr` is now deprecated.

The following *restrictions* apply to `auto_ptrs`:

- the `auto_ptr` class does not support *move semantics*;
- the `auto_ptr` object cannot be used to point to arrays of objects;
- the `auto_ptr` cannot be used as a data type of abstract containers.

No further coverage of the `auto_ptr` class is offered by the **C++** Annotations. Existing software should be modified to use `unique_ptrs`, newly designed software should avoid using `auto_ptr` objects.

18.4 The class 'shared_ptr' (C++0x)

In addition to `std::unique_ptr` the C++0x standard offers a reference counting smart pointer by the name of `std::shared_ptr<Type>`.

The shared pointer will automatically destroy its contents once its reference count has decayed to zero.

Different from `unique_ptrs` `shared_ptrs` support copy constructors and standard overloaded assignment operators. Since `shared_ptrs` share the memory they point at move semantics is not a requirement.

As with `unique_ptr` a `shared_ptr` may refer to a dynamically allocated array.

18.4.1 Defining 'shared_ptr' objects (C++0x)

There are three ways to define `shared_ptr` objects. Each definition contains the usual `<type>` specifier between angle brackets. Concrete examples are given in the coming sections, but an overview of the various possibilities is presented here:

- The basic form initializes an `shared_ptr` object to point to a block of memory allocated by the `new` operator:

  ```
  shared_ptr<type> identifier (new-expression [, deleter]);
  ```

  This form is discussed in section 18.4.2.

- Shared pointers support copy construction. When copy constructing a shared pointer both `shared_ptr` objects point at the same memory, and the object's reference count is incremented.

- Another form initializes an `shared_ptr` object using *move semantics*, either supported by the data type itself or forced using `std::move`:

  ```
  // type must support move semantics or std::move(other) must be used
  shared_ptr<type> identifier(a temporary shared_ptr for type);
  ```

  This form is discussed in section 18.4.3.

- The third form simply creates an `shared_ptr` object that does not point to a particular block of memory:

  ```
  shared_ptr<type> identifier;
  ```

  This form is discussed in section 18.4.4.

18.4.2 Pointing to a newly allocated object (C++0x)

The basic form to initialize an `shared_ptr` object is to provide its constructor with a block of memory allocated by `operator new` operator. The generic form is:

```
shared_ptr<type [, deleter_type]> identifier(new-expression
        [, deleter = deleter_type()]);
```

The second (template) argument (`deleter(_type)`) is optional and may refer to a class/object handling the destruction of the allocated memory. It is used in situations comparable to those encountered with `unique_ptr` (cf. section 18.3.2).

Here is an example initializing an `shared_ptr` pointing to a `string` object:

```
shared_ptr<string> strPtr(new string("Hello world"));
```

Note the use of `operator new` in the above expression. The type specified for the `shared_ptr` is identical to the type used in `new` expression.

All member functions available for objects allocated by the `new` expression can be reached via the `shared_ptr` as if it was a plain pointer to the dynamically allocated object. For example, in the following program the text 'C++' is inserted behind the word '`hello`'. This example also shows the use of a copy constructed shared pointer and it shows that following the copy construction both objects point at the same information:

```
#include <iostream>
#include <memory>
#include <cstring>
using namespace std;

int main()
{
    shared_ptr<string> sp(new string("Hello world"));
```

```
        shared_ptr<string> sp2(sp);

        sp->insert(strlen("Hello "), "C++ ");
        cout << *sp << '\n' <<
                *sp2 << endl;
    }
    /*
        produced output:

        Hello C++ world
        Hello C++ world
    */
```

18.4.3 Initializing from a temporary 'shared_ptr' (C++0x)

A `shared_ptr` may also be initialized by an r-value reference (a temporary value) initialized an returned by a function. E.g.,

```
std::shared_ptr<std::string> &&makeString(char const *text)
{
    return std::move(shared_ptr<string>(new std::string(text)));
}
std::shared_ptr<std::string> shared(makeString("hello world"));
```

18.4.4 Creating a plain 'shared_ptr' (C++0x)

We've already seen the third form to create an `shared_ptr` object: Without arguments an empty `shared_ptr` object is constructed not pointing to a particular block of memory:

```
shared_ptr<type> identifier;
```

In this case the underlying pointer is set to `0` (zero). Like a `std::unique_ptr` a `shared_ptr` object can be compared to `0` to see if it has been initialized. E.g., code like

```
shared_ptr<int> ip;

if (!ip)
    cout << "0-pointer with a shared_ptr object" << endl;
```

will produce the shown text. Alternatively, the member `get()` is available. This member function, as well as the other member functions of the class `shared_ptr` are described in the next section.

18.4.5 Operators and members (C++0x)

The following operators are defined for the class `shared_ptr`:

- `shared_ptr &shared_ptr<Type>operator=(shared_ptr<Type> &other)`:

  This operator reduces the reference count of the left-hand side object, deleting its memory when its count decays to zero, and sets its pointer to the memory pointed at by right-hand side object, incrementing its reference count.

- `shared_ptr &shared_ptr<Type>operator bool() const`:

  This operator returns `false` if the `shared_ptr` does not point to memory (i.e., its `get()` member, see below, returns 0). Otherwise, `true` is returned.

- `Type &shared_ptr<Type>operator*()`:

  This operator returns a reference to the information stored in the `shared_ptr` object. It acts like a normal pointer dereference operator.

- `Type *shared_ptr<Type>operator->()`:

  This operator returns a pointer to the information stored in the `shared_ptr` object. Through this operator members of a stored object can be selected. For example:

```
shared_ptr<string> sp(new string("hello"));

cout << sp->c_str() << endl;
```

The following member functions are defined for `shared_ptr` objects:

- `Type *shared_ptr<Type>::get()`:

  This member does the same as `operator->()`: it returns a pointer to the information stored in the `shared_ptr` object. This pointer can be inspected: if it's zero the `shared_ptr` object does not point to any memory. This member cannot be used to let the `shared_ptr` object point to (another) block of memory.

- `Deleter &shared_ptr<Type>::get_deleter()`:

  This member returns a reference to the deleter class object used by the `shared_ptr`.

- `Type *shared_ptr<Type>::release()`:

  This member returns a pointer to the information stored in the `shared_ptr` object, which loses the memory it pointed at (and changes into a 0-pointer). The member can be used to transfer the information stored in the `shared_ptr` object to a plain `Type` pointer. It is the responsibility of the programmer to delete the memory returned by this member function.

- `void shared_ptr<Type>::reset(Type *)`:

  This member may also be called *without* argument, to delete the memory stored in the `shared_ptr` object, or with a pointer to a dynamically allocated block of memory, which will thereupon be the memory accessed by the `shared_ptr` object. This member function can be used to assign a new block of memory (new content) to an `shared_ptr` object.

- `void shared_ptr<Type>::swap(shared_ptr<Type> &&)`:

  This member is used to swap two identically typed `shared_ptr`s.

- `bool shared_ptr<Type>unique() const`:

  This member returns `true` if the memory is referenced by the current object only and returns `false` otherwise

  .

- `size_t shared_ptr<Type>use_count() const`:

  This member returns the number of objects sharing the memory pointed at by their data pointers.

18.4.6 Class constructors and pointer data members (C++0x)

Now that the shared_ptr's main features have been described, consider the following simple class:

```
// required #includes

class Map
{
    std::map<string, Data> *d_map;
    public:
        Map(char const *filename) throw(std::exception);
};
```

The class's constructor Map() performs the following tasks:

- It allocates a std::map object;
- It opens the file whose name is given as the constructor's argument;
- It reads the file, thereby filling the map.

Of course, it may not be possible to open the file. In that case an appropriate exception is thrown. So, the constructor's implementation will look somewhat like this:

```
Map::Map(char const *fname)
:
    d_map(new std::map<std::string, Data>) throw(std::exception)
{
    ifstream istr(fname);
    if (!istr)
        throw std::exception("can't open the file");
    fillMap(istr);
}
```

What's wrong with this implementation? Its main weakness is that it hosts a potential memory leak. The memory leak only occurs when the exception is actually thrown. In all other cases, the function operates perfectly well. When the exception is thrown, the map has just been dynamically allocated. However, even though the class's destructor will dutifully call delete d_map, the destructor is actually never called, as the destructor will only be called to destroy objects that were constructed completely. Since the constructor terminates in an exception, its associated object is not constructed completely, and therefore that object's destructor is never called.

Shared_ptrs (as well as unique_ptrs, cf. section 18.3) may be used to prevent these kinds of problems. By defining d_map as

```
    std::shared_ptr<std::map<std::string, Data> >
```

it suddenly changes into an object. Now, Map's constructor may safely throw an exception. As d_map is an object itself, its destructor will be called by the time the (however incompletely constructed) Map object goes out of scope.

As a rule of thumb: classes should use shared_ptr or unique_ptr objects, rather than plain pointers for their pointer data members if there's any chance that their constructors will end prematurely in an exception.

18.5 Multi Threading (C++0x)

The C++0x standard adds multi-threading to **C++** through the **C++** standard library.

The Annotations don't aim at introducing the concepts behind multi threading. It is a topic by itself and many good reference sources exist (cf. Nichols, B, *et al.*'s `Pthreads Programming`[1], O'Reilly for a good introduction to multi-threading).

Multi threading facilities are offered through the class `std::thread`. Its constructor and assignment operator accept a function (object) that will handle the thread created by the `thread` object.

Thread synchronization is handled by objects of the class `std::mutex` and condition variables are implemented by the class `std::condition_variable`.

In order to use multi threading in **C++** programs the Gnu `g++` compiler requires the use of the `-pthread` flag. E.g., to compile a multi-threaded program defined in the source file `multi.cc` the compiler is called as follows:

```
g++ --std=c++0x -pthread -Wall multi.cc
```

Threads in **C++** are very much under development. It is likely that in the near future features will be added and possibly redefined. The sections below should therefore be read with this in mind.

18.5.1 The class 'std::thread' (C++0x)

The `std::thread` class implements a new thread. It can be constructed empty (in which case no new thread is started yet) but it may also be given a function or function object, in which case the thread is started immediately.

Alternatively, a function or function object may be *assigned* to an existing `thread` object causing a new thread to be launched using that function (object).

There are several ways to end a launched thread. Currently a thread ends when the function called from the thread object ends. Since the `thread` object not only accepts functions but also function objects as its argument, a *local context* may be passed on to the the thread. Here are two examples of how a thread may be started: in the first example a function is passed on to the thread, in the second example a function object:

```
#include <iostream>
#include <thread>
#include <cstdlib>
using namespace std;

void hello()
{
    cout << "hello world\n";
}

class Zero
{
    int *d_data;
    size_t d_size;
    public:
        Zero(int *data, size_t size)
        :
            d_data(data),
            d_size(size)
        {}
        void operator()(int arg) const
        {
            for (int *ptr = d_data + d_size; ptr-- != d_data; )
```

[1] http://oreilly.com/catalog/

```
                *ptr = 0;
        }
};

int main()
{
    int data[30];
    Zero zero(data, 30);
    int value = 0;
    std::thread t1(zero, value);
    std::thread t2(hello);
};
```

Thread objects do not implement a copy constructor, but a move constructor is provided. Threads may be swapped as well, even if they're actually running a thread. E.g.,

```
std::thread t1(Thread());
std::thread t2(Thread());
t1.swap(t2);
```

According to the current specifications of the `thread` class its constructor should also be able to accept additional arguments (in addition to the function (object) handled by the thread object), but currently that facility does not appears to be very well implemented. Any objects otherwise passed to the function call operator could of course also be passed using separate members or via the object's constructors, which is probably an acceptable makeshift solution until multiple arguments can be passed to the thread constructor itself (using perfect forwarding, cf. section 21.5.2).

The thread's overloaded assigment operator can also be used to start a thread. If the current thread object actually runs a thread it is stopped, and the function (object) assigned to the thread object becomes the new running thread. E.g.,

```
std::thread thr;    // no thread runs from thr yet
thr = Thread();     // a thread is launched
```

Threads (among which the thread represented by `main`) may be forced to wait for another threds's completion by calling the other thread's `join()` member. E.g., in the following example `main` launches two threads and wait for the completion of both:

```
int main()
{
    std::thread t1(Thread());
    std::thread t2(Thread());

    t1.join();      // wait for t1 to complete
    t2.join();      // same, t2
}
```

The thread's member `detach()` can be called to disassociate the current thread from its starting thread. Ending a thread other than ending the activities of the function defining the thread's activities appears currently very well implemented.

18.5.2 Synchronization (mutexes) (C++0x)

C++0x offers a series of mutex classes to protect shared data. Apart from the `std::mutex` class a *std::recursive_mutex* is offered: when called multiple times by the same thread it will increase its lock-count. Before other threads may access the protected data the recursive mutex must be unlocked again

that number of times. In addition `std::timed_mutex` and `recursive_timed_mutex` is offered: their locks will time out after a preset amount of time.

In many situations locks will be released at the end of some action block. To simplify locking additional template classes `std::unique_lock<>` and `std::lock_guard<>` are provided. As their constructor locks the data and their destructor unlocks the data they can be defined as local variables, unlocking their data once their scope terminates. Here is a simple example showing its use; at the end of `safeProcess guard` is destroyed, thereby releasing the lock on `data`:

```
std::mutex dataMutex;
Data data;

void safeProcess()
{
    std::lock_guard<std::mutex> guard(dataMutex);
    process(data);
}
```

A `std::unique_lock` is used similarly, but is used when timeouts must be considered as well:

```
std::timed_mutex dataMutex;
Data data;

void safeProcess()
{
    std::unique_lock<std::timed_mutex>
        guard(dataMutex, std::chrono::milliseconds(3));
    if (guard)
        process(data);
}
```

In the above example `guard` tries to obtain the lock during three milliseconds. If `guard`'s `bool` conversion operator returns `true` the lock was obtained and `data` can be processed safely.

A common cause of problems with multi threaded programs is the occurrence of deadlocks. If a deadlock may occur when two locks are required to process data, but one thread obtains the first lock and another thread obtains the second lock. The C++0x standard defines a generic `std::lock` function that may be used to prevent problems like these. The `std::lock` function can be used to lock multiple mutexes in one atomic action. Here is an example:

```
struct SafeString
{
    std::mutex  d_mutex;
    std::string d_text;
};

void calledByThread(SafeString &first, SafeString &second)
{
    std::unique_lock<std::mutex>                        // 1
            lock_first(first.d_mutex, std::defer_lock);

    std::unique_lock<std::mutex>                        // 2
            lock_second(second.d_mutex, std::defer_lock);

    std::lock(lock_first, lock_second);                 // 3

    safeProcess(first.d_text, second.d_text);
}
```

At 1 and 2 unique_locks are created. Locking is deferred until calling `std::lock` in 3. Having obtained the lock, the two `SafeString` text members can both be safely processed by `calledByThread`.

Another problematic issue with threads involves initialization. If multiple threads are running and only the first thread encountering the initialization code should perform the initialization then this problem should not be solved by mutexes. Although the synchronization is accomplished, it will needlessly be accomplished time and again once the initialization phase has been completed. The C++0x standard offers several ways to perform a proper initialization:

- First, a *constexpr constructor* may be defined. Constexpr constructors are currently not yet supported by the `g++` compiler and its discussion is therefore postponen.
- Second, a static variable defined within a compound statement may be used (e.g., a static local variable within a function body). block scope may be used. In C++ static variables defined within a compund statement are initialized the first time the function is called at the point in the code where the static variable is defined as illustrated by the following example:

```
#include <iostream>

struct Cons
{
    Cons()
    {
        std::cout << "Cons called\n";
    }
};

void called(char const *time)
{
    std::cout << time << "time called() activated\n";
    static Cons cons;
}

int main()
{
    std::cout << "Pre-1\n";
    called("first");
    called("second");
    std::cout << "Pre-2\n";
    Cons cons;
}
/*
    Generated output:

            Pre-1
            firsttime called() activated
            Cons called
            secondtime called() activated
            Pre-2
            Cons called
*/
```

  This feature causes a thread to wait automatically if another thread is still initializing the static data (note that *non-static* data never cause problems, as each non-static local variables have lifes that are completely restricted to their own threads).

- If the above two approaches can't be used. The combined use of `std::call_once` and `std::once_flag` result in the one-time execution of a specified function as illustrated by the next example:

```
std::string *global;
```

```
std::once_flag globalFlag;

void initializeGlobal()
{
    global = new std::string("Hello world (why not?)");
}

void safeUse()
{
    std::call_once(globalFlag, initializeGlobal);
    process(*global);
}
```

Mutexes may be used in C++0x after including the header file `mutex`.

18.5.3 Event handling (condition variables) (C++0x)

Consider a classic producer-consumer case: the producer generates items which are consumed by a consumer. The producer can only produce so many items before its storage capacity has filled up and the client cannot consume more items than the producer has produced.

At some point the producer will therefore have to wait until the client has consumed enough and there's space available in the producer's storage. Similarly, the client will have to wait until the producer has produced some more items.

Locking and polling the amount of available items/storage at fixed time intervals usually isn't a good option as it is in principle a wasteful scheme: threads continue to wait during the full interval even though the condition to continue may already have been met; reducing the interval, on the other hand, isn't an attractive option either as it results in a relative increase of the overhead associated with handling the associated mutexes.

Condition variables may be used to solve these kinds of problems. A thread simply sleeps until it is notified by another thread. In general terms this may be accomplished as follows:

```
producer loop:
    - produce the next item
    - wait until there's room to store the item,
        then reduce the available room
    - store the item
    - increment the number of items in store

consumer loop:
    - wait until there's an item in store,
        then reduce the number of items in store
    - remove the item from the store
    - increment the number of available storage locations
    - do something with the retrieved item
```

It is important that the two storage administrative tasks (registering the number of available items and available storage locations) are either performed by the client or by the producer. 'Waiting' in this case means:

- Get a lock on the variable containing the actual count
- As long as the count is zero: wait, relase the lock until another thread has increased the count, re-acquire the lock.
- Reduce the count

- Release the lock.

To implement this a `condition_variable` is used. The variable containing the actual count is called `semaphore`; and it can be protected by a `mutex sem_mutex`. In addition a `condition_variable condition` is defined. The waiting process is defined in the following function `down` implemented as follows:

```
void down()
{
    unique_lock<mutex> lk(sem_mutex);   // get the lock
    while (semaphore == 0)
        condition.wait(lk);             // see 1, below.
    --semaphore;                        // dec. semaphore count
}                                       // the lock is released
```

At 1 the condition variable's `wait()` member internally releases the lock, wait for a notification to continue, and re-acquires the lock just before returning. Consequently, the `down`'s code always has complete and unique control over `semaphore`.

What about notifying the condition variable? This is handled by the 'increment the number ...' parts in the abovementioned produced and consumer loops. These parts are defined by the following `up()` function, implemented as follows:

```
void up()
{
    lock_guard<std::mutex> lk(sem_mutex);   // get the lock
    if (semaphore++ == 0)
        condition.notify_one();             // see 2, below
}                                           // the lock is released
```

At 2 `semaphore` is always incremented. However, by using a postfix increment it can be tested for being zero at the same time and if it was zero, initially then `semaphore` is now one. Consequently, the thread waiting for `semaphore` being unequal to zero may now continue. `Condition.notify_one()` will notify a waiting thread (see `down`'s implementation above). If situations where multiple threads are waiting '`notify_all()`' can be used.

Handling `semaphore` can very well be implemented in a class `Semaphore`, offering members `down()` and `up()`. For a more extensive discussion of semaphores see Tanenbaum, A.S. (2006) Structured Computer Organization, Pearson Prentice-Hall.

Using the semaphore facilities, embedded in a class `Semaphore` whose constructor expects an initial value of its `semaphore` data member, the classic producer and consumer case can now easily be implemented in the following multi-threaded program[2]:

```
Semaphore available(10);
Semaphore filled(0);
std::queue itemQueue;

void producer()
{
    size_t item = 0;
    while (true)
    {
        ++item;
        available.down();
        itemQueue.push(item);
```

[2] A more elaborate example of the producer-consumer program is found in the `yo/stl/examples/events.cc` file in the **C++** Annotations's source archive

```
            filled.up();
        }
    }
    void client()
    {
        while (true)
        {
            filled.down();
            size_t item = itemQueue.front();
            itemQueue.pop();
            available.up();
            process(item);      // not implemented here
        }
    }

    int main()
    {
        thread produce(producer);
        thread consume(consumer);

        produce.join();
        return 0;
    }
```

To use `condition_variable` objects the header file `condition_variable` must be included.

18.6 Lambda functions (C++0x)

The C++0x standard introduces *lambda functions* into **C++**. As there is currently no indication as to when this feature will become available in the `g++` compiler this section will only cover the concept, the proposed syntax and some examples.

As we've seen the generic algorithms often accept an argument which can either be a function object or a plain function. Examples are the `sort` and `find_if` generic algorithms. When the function called by the generic algorithm must remember its state a function object is appropriate, otherwise a plain function will do.

The function or function object is usually not readily available. Often it must be defined in or near the function using the generic algorithm. Often the software engineer will resort to anonymous namespaces in which a class or function is defined that is thereupon used in the function calling the generic algorithm. If that function is itself a member function the need may be felt to access other members of its class from the function called by the generic algorithm. Often this results in a significant amount of code (defining the class), in complex code (to make available software elements that aren't native to the called function (object)), and -at the level of the source file- code that is irrelevant at the current level of specification. Nested classes don't solve these problems and, moreover, nested classes can't be used in templates.

Solutions to the above problems exist. In section 23.8 a general approach is discussed allowing the use of members and/or local variables in the context of a function or function object called from generic algorithms.

However, the solutions given in section 23.8 are in fact makeshift solutions that need to be used as long as the language doesn't offer lambda functions. A lambda function is an anonymous function. Such a function may be defined on the spot, and will exist only during the lifetime of the statement of which it is a part.

Here is an example of the definition of a lambda function:

```
[](int x, int y)
{
    return x * y;
}
```

This particular function expects two `int` arguments and returns their product. It could be used e.g., in combination with the `accumulate` generic algorithm to compute the product of a series of `int` values stored in a vector:

```
cout << accumulate(vi.begin(), vi.end(), 1,
            [](int x, int y) { return x * y; });
```

The above lambda function will use the implicit return ih(return type: implicit) type `decltype(x * y)`. An implicit return type can only be used if the lambda function has a single statement of the form `return expression;`.

Alternatively, the return type can be explicitly specified using a late-specified return type, (cf. section 3.3.5):

```
[](int x, int y) -> int
{
    int z = x + y;
    return z + x;
}
```

A lambda function not returning a value (i.e., a void-function) does not have to specify a return type at all.

Variables having the same scope as the lambda function can be accessed from the lambda function using references. This allows passing the local context to a lambda function. Such variables are called a *closure*. Here is an example:

```
void showSum(vector<int> &vi)
{
    int total = 0;
    for_each(
        vi.begin(), vi.end(),
        [&total](int x)
        {
            total += x;
        }
    );
    std::cout << total;
}
```

The variable `int total` is passed to the lambda function as a reference (`[&total]`) and can directly be accessed by the function. Its parameter list merely defines an `int x`, which is initialized in turn by each of the values stored in `vi`. Once the generic algorithm has completed `showSum`'s variable `total` has received a value equal to the sum of all the vector's values. It has outlived the lambda function and its value is displayed.

If a closure variable is defined without the reference symbol it becomes a simple value which is initialized by the local variable when the lambda function is passed to the generic algorithm. Usually closure variables are passed by reference. If *all* local variables are to be passed by reference the notation `[&]` can be used (to pass the full closure by value use `[=]`):

```
void show(vector<int> &vi)
```

```
    {
        int sum = 0;
        int prod = 1;
        for_each(
            vi.begin(), vi.end(),
            [&](int x)
            {
                sum += x;
                prod *= x;
            }
        );
        std::cout << sum << ' ' << prod;
    }
```

Combinations of variables passed by value, but some by reference or passed by reference but some by value is possible. In that case the default is followed by a list of variables passed differently. E.g.,

```
    [&, value](int x)
    {
        total +=  x * value;
    };
```

In this case `total` will be passed by reference, `value` by value.

Within a class context, members may define lambda functions as well. In those cases the lambda function has full access to all the class's members. In other words, it is defined as a friend of the class. For example:

```
    class Data
    {
        std::vector<std::string> d_names;

        public:
            void show() const;
            {
                int count = 0;
                std::for_each(d_names.begin(), d_end(),
                    [this, &count](std::string const &name)
                    {
                        std::cout << ++count <<
                            this->capitalized(name) << endl;
                    }
                )
            }
        private:
            std::string capitalized(std::string const &name);
    }
```

Note the use of the `this` pointer: inside the lambda function it must explicitly be used (so, `this->capitalized(name)` is used rather than `capitalized(name)`). In addition, in situations like the above `this` will automatically be available when `[&]` or `[=]` is used. So, the above lambda function could have been defined as:

```
                    [&](std::string const &name)
                    {
                        std::cout << ++count <<
                            this->capitalized(name) << endl;
                    }
```

When lambda functions are compared to, e.g., the function wrappers discussed in section 23.8 it probably won't come as a surprise that lambda functions are function objects. It is possible to assign a lambda function to a variable. An example of such an assignment (using `auto` to define the type of the variable) is:

```
auto lambdaFun = [this]()
                 {
                    this->member();
                 };
```

Lambda functions are not yet supported by the `g++` compiler.

18.7 Polymorphous wrappers for function objects (C++-0x)

In **C++** (member) function pointers have fairly strict targets: these pointers may only point to (member) functions of their (class) scope. However when defining templates the (class) type of the function pointer may vary from instantiation to instantiation.

To accomodate for these situations the C++0x standard introduces *polymorphous (function object) wrappers*. Polymorphous wrappers can refer to function pointers, member functions or functors, as long as they match in type and number of their parameters.

Polymorphous wrappers are not yet fully supported by the `g++` compiler.

18.8 Randomization and Mathematical Distributions (C++0x)

The C++0x standard introduces several standard mathematical (statistical) distributions into the STL. These distributions allow programmers to obtain randomly selected values from a specified distribution.

Using these distrbutions is based on new randum number generator functions, differing from the traditional **rand**(3) function provided by the **C** standard library. These random number generators are used to produce pseudo-random numbers, which are then filtered through a particular distribution to obtain values that are randomly selected from the specified distributino.

18.8.1 Random Number Generators (C++0x)

The following generators are available:

Class template	Integral/Floating point	Quality	Speed	Size of state
`linear_congruential`	Integral	Medium	Medium	1
`subtract_with_carry`	Both	Medium	Fast	25
mersenne_twister	Integral	Good	Fast	624

The `linear_congruential` random number generator computes

$$\texttt{value}_{i+1} = \texttt{a} * \texttt{value}_i + \texttt{c} \ \% \ \texttt{m}$$

Its template arguments are the data type to contain the generated random values, the multiplier `a`, the additive constant `c` and the modulo value `m`. E.g.,

```
linear_congruential<int, 10, 3, 13> lc;
```

The `linear_congruential` generator may also be seeded by providing its constructor with a seeding-argument. E.g., `lc(time(0))`.

The `subtract_with_carry` random number generator computes

$$\texttt{value}_i \texttt{ = value}_{i-s} \texttt{ - value}_{i-r} \texttt{ - carry}_{i-1} \texttt{ \% m}$$

Its template arguments are the data type to contain the generated random values, the modulo value `m`, the subtractive constants `s` and `r`, respectively. E.g.,

```
subtract_with_carry<int, 13, 3, 13> sc;
```

The `subtract_with_carry` generator may also be seeded by providing its constructor with a seeding-argument. E.g., `sc(time(0))`.

The predefined `mersenne_twister mt19937` (predefined using a `typedef` defined by the `random` header file) is used in the examples below. It can be constructed using `std::mt19937 mt` or it can be seeded using an argument (e.g., `std::mt19937 mt(time(0))`). Other ways to initialize the `mersenne_twister` are byond the scope of the **C++** Annotations (but see Lewis *et al.*[3] (1969)).

The random number generators may also be seeded by calling their members `seed()` which accepts an `unsigned long` or a generator function (e.g., `lc.seed(lc), lc.seed(mt)`).

The random number generators implement members `min()` and `max()` returning, respectively, their minimum and maximum values (inclusive). If a reduced range is required the generators can be nested in a function or class changing its range.

18.8.2 Mathematical distributions (C++0x)

The following standard distributions are currently available (an example showing their use is provided at the end of this section). The reader is referred to, e.g., for the *Bernoulli distribution* `http://en.wikipedia.org/wiki/B` for more information about the distributions mentioned below.

All distributions offer the following members (*result_type* being the type name of the values returned by the distribution, *distribution-name* being the type name of the mathematical distribution, e.g., `bernoulli_distribution`):

- `result_type operator()(RandomNumberGenerator)`: The next randomly generated value is returned.
- `std::istream &operator«(std::istream &in, distribution-name &object)`: A parameters of the distribution are extracted from an `std::istream`;
- `std::ostream &operator«(std::ostream &out, bernoulli_distribution const &bd)`: A parameters of the distribution are inserted into an `std::ostream`

Distributions:

- `bernoulli_distribution`
  Logical `true` values are generated with a certain probability `p`.
  Constructor:
  - `bernoulli_distribution(double p = 0.5)`

[3] Lewis, P.A.W., Goodman, A.S., and Miller, J.M. (1969), A pseudorandom number generator for the System/360, IBM Systems Journal, 8, 136-146.

- `binomial_distribution<IntType = int, RealType = double>`
  `IntType`: the type of the generated random value (by default `int`).
  `RealType`: the probability parameter of the binomial distribution.
  Random integral values are generated according to the specified binomial distributionnl() Constructor:

  - `binomial_distribution<IntType, RealType>(IntType t, RealType p = RealType(0.5))`

- `gamma_distribution<Type = double>`
  `Type`: the parameter of the gamma distribution.
  Random values are generated according to the specified gamma distribution.
  Constructor:

  - `gamma_distribution<IntType, Type>(Type alpha = Type(1))`

- `geometric_distribution<IntType = int, RealType = double>`
  `IntType`: the type of the generated random value (by default `int`).
  `RealType`: the probability parameter of the geometric distribution.
  Random integral values are generated according to the specified geometric distributionnl() Constructor:

  - `geometric_distribution<IntType, RealType>(RealType p = RealType(0.5))`

- `exponential_distribution<Type = double>`
  `Type`: the parameter of the exponential distribution.
  Random integral values are generated according to the specified exponential distributionnl() Constructor:

  - `exponential_distribution<IntType, Type>(Type lambda = Type(1))`

- `normal_distribution<Type = double>`
  `Type`: the type of the generated random value (by default `double`).
  Constructor:

  - `normal_distribution<Type>(ReadType mean = Type(0), Type sigma = Type(1))`

- `poisson_distribution<IntType = int, RealType = double>`
  `IntType`: the type of the generated random value (by default `int`).
  `RealType`: the type of the parameter of the distribution (by default `double`).
  Random integral values are generated that are distributed according to a poisson distribution with parameter `mean`. Constructor:

  - `poisson_distribution<IntType, RealType>(RealType mean = RealType(1))`

- `uniform_int<Type = int>`
  `Type`: the type of the generated random value (by default `int`).
  Random integral values are generated that are uniformly distributed over a certain range.
  Constructor:

  - `uniform_int<Type>(Type min = 0, Type max = 9)`

  Additional member:

  - `Type operator()(RandomNumberGenerator), Type upper`
    Random numbers are generated in the range ttrange(0)(upper) (half-inclusive).

- `uniform_real<Type = real>`
  `Type`: the type of the generated random value (by default `real`).
  Random real values are generated that are uniformly distributed over a certain range.
  Constructor:
  - `uniform_real<Type>(Type min = Type(0), Type max = Type(1))`

  Note: the current implementation generates

  ```
  GRN * (max - min) + min
  ```

  which might not be what you'd expect.

Some of the distributions mentioned below appear not yet to be fully operational in the STL. This appears to be the case with the `binomial_distribution`, the `gamma_distribution`, the `normal_distribution`, and the `poisson_distribution`. These distributions will likely become fulle operational in the near future.

Here is an example showing the outcome of a statistical experiment consisting of throwing an honest coin 20 times:

```
#include <iostream>
#include <random>
#include <ctime>
using namespace std;

int main()
{
    std::mt19937 mt(time(0));
    bernoulli_distribution bd;

    for (int idx = 0; idx < 20; ++idx)
        cout << (bd(mt) ? "heads" : "tail") << (idx + 1 == 10 ? '\n' : ' ');
    cout << endl;
}
```

Chapter 19

The STL Generic Algorithms

19.1 The Generic Algorithms

In the previous chapter the Standard Template Library (STL) was introduced. An important element of the STL, the *generic algorithms*, was not covered in that chapter as it covers a fairly extensive part of the STL. Over time the STL has grown considerably, mainly as a result of a growing importance and appreciation of *templates*. Covering generic algorithm in the STL chapter itself would turn that chapter into an unwieldy one and so the generic algoritmhs were moved to a chapter of their own.

Generic algorithms perform an amazing task: due to the strenght of templates algorithms could be developed that can be applied to a wide range of different data types while maintaining type safety. The prototypical example of this is the `sort` generic algorithm (cf. section 19.1.58). To contrast: while **C** requires programmers to write callback functions in which type-unsafe `void const *` parameters have to be used, internally forcing the programmer to use casts, the STL `sort` algorithm allows the programmer in many cases to state something akin to

```
sort(first-element, last-element)
```

to sort a series of elements in a type-safe way.

Generic algoritms should be used wherever possible. Avoid the urge to design your own code for commonly encountered algorithms. Make it a habit to *first* thoroughly search the generic algorithms for an available candidate. The generic algorithms should become your *weapon of choice* when writing code: acquire full familiarity with them and turn their use into your 'second nature'.

The following sections describe the STL's generic algorithms in alphabetical order. For each algorithm the following information is provided:

- The required header file;
- The function prototype;
- A short description;
- A short example.

In the prototypes of the algorithms `Type` is used to specify a generic data type. Also, the particular type of iterator (see section 18.2) that is required is mentioned, as well as other generic types that might be required (e.g., performing `BinaryOperations`, like `plus<Type>()`).

Almost every generic algorithm expects an iterator range `[first, last)`, defining the range of elements on which the algorithm operates. The iterators point to objects or values. When an iterator points to a `Type` value or object, function objects used by the algorithms usually receive `Type const`

& objects or values: function objects can therefore not modify the objects they receive as their arguments. This does not hold true for *modifying generic algorithms*, which *are* (of course) able to modify the objects they operate upon.

Generic algorithms may be categorized. In the **C++** Annotations the following categories of generic algorithms are distinguished:

- Comparators: comparing (ranges of) elements:

  Requires: `#include <algorithm>`
  equal(); includes(); lexicographical_compare(); max(); min(); mismatch();

- Copiers: performing copy operations:

  Requires: `#include <algorithm>`
  copy(); copy_backward(); partial_sort_copy(); remove_copy(); remove_copy_if(); replace_copy(); replace_copy_if(); reverse_copy(); rotate_copy(); unique_copy();

- Counters: performing count operations:

  Requires: `#include <algorithm>`
  count(); count_if();

- Heap operators: manipulating a max-heap:

  Requires: `#include <algorithm>`
  make_heap(); pop_heap(); push_heap(); sort_heap();

- Initializers: initializing data:

  Requires: `#include <algorithm>`
  fill(); fill_n(); generate(); generate_n();

- Operators: performing arithmetic operations of some sort:

  Requires: `#include <numeric>`
  accumulate(); adjacent_difference(); inner_product(); partial_sum();

- Searchers: performing search (and find) operations:

  Requires: `#include <algorithm>`
  adjacent_find(); binary_search(); equal_range(); find(); find_end(); find_first_of(); find_if(); lower_bound(); max_element(); min_element(); search(); search_n(); set_difference(); set_intersection(); set_symmetric_difference(); set_union(); upper_bound();

- Shufflers: performing reordering operations (sorting, merging, permuting, shuffling, swapping):

  Requires: `#include <algorithm>`
  inplace_merge(); iter_swap(); merge(); next_permutation(); nth_element(); partial_sort(); partial_sort_copy(); partition(); prev_permutation(); random_shuffle(); remove(); remove_copy(); remove_copy_if(); remove_if(); reverse(); reverse_copy(); rotate(); rotate_copy(); sort(); stable_partition(); stable_sort(); swap(); unique();

- Visitors: visiting elements in a range:

  Requires: `#include <algorithm>`
  for_each(); replace(); replace_copy(); replace_copy_if(); replace_if(); transform(); unique_copy();

19.1.1 accumulate()

- Header file:

```
#include <numeric>
```

- Function prototypes:
  - `Type accumulate(InputIterator first, InputIterator last, Type init);`
  - `Type accumulate(InputIterator first, InputIterator last, Type init, BinaryOperation op);`
- Description:
  - The first prototype: `operator+()` is applied to all elements implied by the iterator range and to the initial value `init`. The resulting value is returned.
  - The second prototype: the binary operator `op()` is applied to all elements implied by the iterator range and to the initial value `init`, and the resulting value is returned.
- Example:

```
#include <numeric>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    int         ia[] = {1, 2, 3, 4};
    vector<int> iv(ia, ia + 4);

    cout <<
        "Sum of values: " << accumulate(iv.begin(), iv.end(), int()) <<
        endl <<
        "Product of values: " << accumulate(iv.begin(), iv.end(), int(1),
                                            multiplies<int>()) << endl;

    return 0;
}
/*
    Generated output:

    Sum of values: 10
    Product of values: 24
*/
```

19.1.2 adjacent_difference()

- Header file:

```
#include <numeric>
```

- Function prototypes:
  - `OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);`
  - `OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);`

- Description: All operations are performed on the original values, all computed values are returned values.
  - The first prototype: the first returned element is equal to the first element of the input range. The remaining returned elements are equal to the difference of the corresponding element in the input range and its previous element.
  - The second prototype: the first returned element is equal to the first element of the input range. The remaining returned elements are equal to the result of the binary operator `op` applied to the corresponding element in the input range (left operand) and its previous element (right operand).
- Example:

```
#include <numeric>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    int             ia[] = {1, 2, 5, 10};
    vector<int>     iv(ia, ia + 4);
    vector<int>     ov(iv.size());

    adjacent_difference(iv.begin(), iv.end(), ov.begin());

    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    adjacent_difference(iv.begin(), iv.end(), ov.begin(), minus<int>());

    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    generated output:

    1 1 3 5
    1 1 3 5
*/
```

19.1.3 adjacent_find()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);`
  - `OutputIterator adjacent_find(ForwardIterator first, ForwardIterator last, Predicate pred);`
- Description:
  - The first prototype: the iterator pointing to the first element of the first pair of two adjacent equal elements is returned. If no such element exists, `last` is returned.

  - The second prototype: the iterator pointing to the first element of the first pair of two adjacent elements for which the binary predicate `pred` returns `true` is returned. If no such element exists, `last` is returned.

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>

class SquaresDiff
{
    size_t d_minimum;

    public:
        SquaresDiff(size_t minimum)
        :
            d_minimum(minimum)
        {}
        bool operator()(size_t first, size_t second)
        {
            return second * second - first * first  >= d_minimum;
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "Alpha", "bravo", "charley", "delta", "echo", "echo",
            "foxtrot", "golf"
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    string *result = adjacent_find(sarr, last);

    cout << *result << endl;
    result = adjacent_find(++result, last);

    cout << "Second time, starting from the next position:\n" <<
        (
            result == last ?
                "** No more adjacent equal elements **"
            :
                "*result"
        ) << endl;

    size_t iv[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    size_t *ilast = iv + sizeof(iv) / sizeof(size_t);
    size_t *ires = adjacent_find(iv, ilast, SquaresDiff(10));

    cout <<
        "The first numbers for which the squares differ at least 10: "
        << *ires << " and " << *(ires + 1) << endl;

    return 0;
}
/*
    Generated output:
```

```
    echo
    Second time, starting from the next position:
    ** No more adjacent equal elements **
    The first numbers for which the squares differ at least 10: 5 and 6
*/
```

19.1.4 binary_search()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `bool binary_search(ForwardIterator first, ForwardIterator last, Type const &value);`
  - `bool binary_search(ForwardIterator first, ForwardIterator last, Type const &value, Comparator comp);`
- Description:
  - The first prototype: `value` is looked up using binary search in the range of elements implied by the iterator range `[first, last)`. The elements in the range must have been sorted by the `Type::operator<()` function. `True` is returned if the element was found, `false` otherwise.
  - The second prototype: `value` is looked up using binary search in the range of elements implied by the iterator range `[first, last)`. The elements in the range must have been sorted by the `Comparator` function object. `True` is returned if the element was found, `false` otherwise.
- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <functional>
using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel"
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    bool result = binary_search(sarr, last, "foxtrot");

    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;

    reverse(sarr, last);                // reverse the order of elements
                                        // binary search now fails:
    result = binary_search(sarr, last, "foxtrot");
    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;
                                        // ok when using appropriate
                                        // comparator:
    result = binary_search(sarr, last, "foxtrot", greater<string>());
    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;
```

```
    return 0;
}
/*
    Generated output:

    found foxtrot
    didn't find foxtrot
    found foxtrot
*/
```

19.1.5 copy()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `OutputIterator copy(InputIterator first, InputIterator last, OutputIterator destination);`
- Description:
  - The range of elements implied by the iterator range `[first, last)` is copied to an output range, starting at `destination` using the assignment operator of the underlying data type. The return value is the OutputIterator pointing just beyond the last element that was copied to the destination range (so, 'last' in the destination range is returned).
- Example:
  Note the second call to `copy()`. It uses an `ostream_iterator` for `string` objects. This iterator will write the `string` values to the specified `ostream` (i.e., `cout`), separating the values by the specified separation string (i.e., `" "`).

```
#include <algorithm>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel"
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy(sarr + 2, last, sarr); // move all elements two positions left

                                // copy to cout using an ostream_iterator
                                // for strings,
    copy(sarr, last, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
```

```
    Generated output:

    charley delta echo foxtrot golf hotel golf hotel
*/
```

- See also: `unique_copy()`

19.1.6 copy_backward()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `BidirectionalIterator copy_backward(InputIterator first, InputIterator last, BidirectionalIterator last2);`
- Description:
  - The range of elements implied by the iterator range `[first, last)` are copied from the element at position `last - 1` until (and including) the element at position `first` to the element range, *ending* at position `last2 - 1` using the assignment operator of the underlying data type. The destination range is therefore `[last2 - (last - first), last2)`.
    The return value is the BidirectionalIterator pointing to the last element that was copied to the destination range (so, 'first' in the destination range, pointed to by `last2 - (last - first)`, is returned).
- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel"
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        copy_backward(sarr + 3, last, last - 3),
        last,
        ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Generated output:

    golf hotel foxtrot golf hotel foxtrot golf hotel
*/
```

19.1.7 count()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `size_t count(InputIterator first, InputIterator last, Type const &value);`
- Description:
  - The number of times `value` occurs in the iterator range `[first, last)` is returned. Use `Type::operator==()` to determine whether `value` is equal to an element in the iterator range.
- Example:

```
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};

    cout << "Number of times the value 3 is available: " <<
        count(ia, ia + sizeof(ia) / sizeof(int), 3) <<
        endl;

    return 0;
}
/*
    Generated output:

    Number of times the value 3 is available: 3
*/
```

19.1.8 count_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `size_t count_if(InputIterator first, InputIterator last, Predicate predicate);`
- Description:
  - The number of times unary predicate '`predicate`' returns `true` when applied to the elements implied by the iterator range `[first, last)` is returned.
- Example:

```
#include <algorithm>
#include <iostream>

class Odd
{
```

```
    public:
        bool operator()(int value)
        {
            return value & 1;
        }
};

using namespace std;

int main()
{
    int     ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};

    cout << "The number of odd values in the array is: " <<
        count_if(ia, ia + sizeof(ia) / sizeof(int), Odd()) << endl;

    return 0;
}
/*
    Generated output:

    The number of odd values in the array is: 5
*/
```

19.1.9 equal()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `bool equal(InputIterator first, InputIterator last, InputIterator otherFirst);`
  - `bool equal(InputIterator first, InputIterator last, InputIterator otherFirst, BinaryPredicate pred);`
- Description:
  - The first prototype: the elements in the range `[first, last)` are compared to a range of equal length starting at `otherFirst`. The function returns `true` if the visited elements in both ranges are equal pairwise. The ranges need not be of equal length, only the elements in the indicated range are considered (and must be available).
  - The second prototype: the elements in the range `[first, last)` are compared to a range of equal length starting at `otherFirst`. The function returns `true` if the binary predicate, applied to all corresponding elements in both ranges returns `true` for every pair of corresponding elements. The ranges need not be of equal length, only the elements in the indicated range are considered (and must be available).
- Example:

```
#include <algorithm>
#include <string>
#include <iostream>

class CaseString
{
    public:
        bool operator()(std::string const &first,
```

```
                            std::string const &second) const
        {
            return !strcasecmp(first.c_str(), second.c_str());
        }
};

using namespace std;

int main()
{
    string first[] =
        {
            "Alpha", "bravo", "Charley", "delta", "Echo",
            "foxtrot", "Golf", "hotel"
        };
    string second[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel"
        };
    string *last = first + sizeof(first) / sizeof(string);

    cout << "The elements of ‘first’ and ‘second’ are pairwise " <<
        (equal(first, last, second) ? "equal" : "not equal") <<
        endl <<
        "compared case-insensitively, they are " <<
        (
            equal(first, last, second, CaseString()) ?
                "equal" : "not equal"
        ) << endl;

    return 0;
}
/*
    Generated output:

    The elements of ‘first’ and ‘second’ are pairwise not equal
    compared case-insensitively, they are equal
*/
```

19.1.10 equal_range()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, Type const &value);`
  - `pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, Type const &value, Compare comp);`
- Description (see also identically named member functions of, e.g., the `map` (section 12.3.6) and `multimap` (section 12.3.7)):
  - The first prototype: starting from a sorted sequence (where the `operator<()` of the data type to which the iterators point was used to sort the elements in the provided range), a pair of iterators is returned representing the return value of, respectively, `lower_bound()`

(returning the first element that is not smaller than the provided reference value, see section 19.1.25) and `upper_bound()` (returning the first element beyond the provided reference value, see section 19.1.66).

- The second prototype: starting from a sorted sequence (where the `comp` function object was used to sort the elements in the provided range), a pair of iterators is returned representing the return values of, respectively, the functions `lower_bound()` (section 19.1.25) and `upper_bound()` (section 19.1.66).

- Example:

```
#include <algorithm>
#include <functional>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    int                 range[] = {1, 3, 5, 7, 7, 9, 9, 9};
    size_t const      size = sizeof(range) / sizeof(int);

    pair<int *, int *>  pi;

    pi = equal_range(range, range + size, 6);

    cout << "Lower bound for 6: " << *pi.first << endl;
    cout << "Upper bound for 6: " << *pi.second << endl;

    pi = equal_range(range, range + size, 7);

    cout << "Lower bound for 7: ";
    copy(pi.first, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Upper bound for 7: ";
    copy(pi.second, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    sort(range, range + size, greater<int>());

    cout << "Sorted in descending order\n";

    copy(range, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    pi = equal_range(range, range + size, 7, greater<int>());

    cout << "Lower bound for 7: ";
    copy(pi.first, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Upper bound for 7: ";
    copy(pi.second, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:
```

```
                Lower bound for 6: 7
                Upper bound for 6: 7
                Lower bound for 7: 7 7 9 9 9
                Upper bound for 7: 9 9 9
                Sorted in descending order
                9 9 9 7 7 5 3 1
                Lower bound for 7: 7 7 5 3 1
                Upper bound for 7: 5 3 1
    */
```

19.1.11 fill()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void fill(ForwardIterator first, ForwardIterator last, Type const &value);`
- Description:
  - all the elements implied by the iterator range `[first, last)` are initialized to `value`, overwriting the previous stored values.
- Example:

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    vector<int>     iv(8);

    fill(iv.begin(), iv.end(), 8);

    copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    8 8 8 8 8 8 8 8
*/
```

19.1.12 fill_n()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void fill_n(ForwardIterator first, Size n, Type const &value);`
- Description:
  - `n` elements starting at the element pointed to by `first` are initialized to `value`, overwriting the previous stored values.
- Example:

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    vector<int>     iv(8);

    fill_n(iv.begin() + 2, 4, 8);

    copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    0 0 8 8 8 8 0 0
*/
```

19.1.13 find()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `InputIterator find(InputIterator first, InputIterator last, Type const &value);`
- Description:
  - Element `value` is searched for in the range of the elements implied by the iterator range `[first, last)`. An iterator pointing to the first element found is returned. If the element was not found, `last` is returned. The `operator==()` of the underlying data type is used to compare the elements.
- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
```

```
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo"
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find(sarr, last, "delta"), last, ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    if (find(sarr, last, "india") == last)
    {
        cout << "'india' was not found in the range\n";
        copy(sarr, last, ostream_iterator<string>(cout, " "));
        cout << endl;
    }

    return 0;

}
/*
    Generated output:

    delta echo
    'india' was not found in the range
    alpha bravo charley delta echo
*/
```

19.1.14 find_end()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)`
  - `ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)`
- Description:
  - The first prototype: the sequence of elements implied by `[first1, last1)` is searched for the last occurrence of the sequence of elements implied by `[first2, last2)`. If the sequence `[first2, last2)` is not found, `last1` is returned, otherwise an iterator pointing to the first element of the matching sequence is returned. The `operator==()` of the underlying data type is used to compare the elements in the two sequences.
  - The second prototype: the sequence of elements implied by `[first1, last1)` is searched for the last occurrence of the sequence of elements implied by `[first2, last2)`. If the sequence `[first2, last2)` is not found, `last1` is returned, otherwise an iterator pointing to the first element of the matching sequence is returned. The provided binary predicate is used to compare the elements in the two sequences.
- Example:

```
#include <algorithm>
```

```
#include <string>
#include <iterator>
#include <iostream>

class Twice
{
    public:
        bool operator()(size_t first, size_t second) const
        {
            return first == (second << 1);
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel",
            "foxtrot", "golf", "hotel",
            "india", "juliet", "kilo"
        };
    string search[] =
        {
            "foxtrot",
            "golf",
            "hotel"
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find_end(sarr, last, search, search + 3),   // sequence starting
        last, ostream_iterator<string>(cout, " ")   // at 2nd 'foxtrot'
    );
    cout << endl;

    size_t range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10};
    size_t nrs[]   = {2, 3, 4};

    copy                // sequence of values starting at last sequence
    (                   // of range[] that are twice the values in nrs[]
        find_end(range, range + 9, nrs, nrs + 3, Twice()),
        range + 9, ostream_iterator<size_t>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Generated output:

    foxtrot golf hotel india juliet kilo
    4 6 8 10
*/
```

19.1.15 find_first_of()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)`
  - `ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)`
- Description:
  - The first prototype: the sequence of elements implied by `[first1, last1)` is searched for the first occurrence of an element in the sequence of elements implied by `[first2, last2)`. If no element in the sequence `[first2, last2)` is found, `last1` is returned, otherwise an iterator pointing to the first element in `[first1, last1)` that is equal to an element in `[first2, last2)` is returned. The `operator==()` of the underlying data type is used to compare the elements in the two sequences.
  - The second prototype: the sequence of elements implied by `[first1, first1)` is searched for the first occurrence of an element in the sequence of elements implied by `[first2, last2)`. Each element in the range `[first1, last1)` is compared to each element in the range `[first2, last2)`, and an iterator to the first element in `[first1, last1)` for which the binary predicate `pred` (receiving an the element out of the range `[first1, last1)` and an element from the range `[first2, last2)`) returns `true` is returned. Otherwise, `last1` is returned.
- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

class Twice
{
    public:
        bool operator()(size_t first, size_t second) const
        {
            return first == (second << 1);
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel",
            "foxtrot", "golf", "hotel",
            "india", "juliet", "kilo"
        };
    string search[] =
        {
            "foxtrot",
```

```
            "golf",
            "hotel"
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (                                                   // sequence starting
        find_first_of(sarr, last, search, search + 3),  // at 1st 'foxtrot'
        last, ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    size_t range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10};
    size_t nrs[]   = {2, 3, 4};

    copy            // sequence of values starting at first sequence
    (               // of range[] that are twice the values in nrs[]
        find_first_of(range, range + 9, nrs, nrs + 3, Twice()),
        range + 9, ostream_iterator<size_t>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Generated output:

    foxtrot golf hotel foxtrot golf hotel india juliet kilo
    4 6 8 10 4 6 8 10
*/
```

19.1.16 find_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);`
- Description:
  - An iterator pointing to the first element in the range implied by the iterator range `[first, last)` for which the (unary) predicate `pred` returns `true` is returned. If the element was not found, `last` is returned.
- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

class CaseName
{
    std::string  d_string;
```

```
        public:
            CaseName(char const *str): d_string(str)
            {}
            bool operator()(std::string const &element)
            {
                return !strcasecmp(element.c_str(), d_string.c_str());
            }
    };

    using namespace std;

    int main()
    {
        string sarr[] =
            {
                "Alpha", "Bravo", "Charley", "Delta", "Echo"
            };
        string  *last = sarr + sizeof(sarr) / sizeof(string);

        copy
        (
            find_if(sarr, last, CaseName("charley")),
            last, ostream_iterator<string>(cout, " ")
        );
        cout << endl;

        if (find_if(sarr, last, CaseName("india")) == last)
        {
            cout << "'india' was not found in the range\n";
            copy(sarr, last, ostream_iterator<string>(cout, " "));
            cout << endl;
        }

        return 0;

    }
    /*
        Generated output:

        Charley Delta Echo
        'india' was not found in the range
        Alpha Bravo Charley Delta Echo
    */
```

19.1.17 for_each()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `Function for_each(ForwardIterator first, ForwardIterator last, Function func);`
- Description:
  - Each of the elements implied by the iterator range `[first, last)` is passed in turn as a reference to the function (or function object) `func`. The function may modify the elements

it receives (as the used iterator is a forward iterator). Alternatively, if the elements should be transformed, `transform()` (see section 19.1.63) can be used. The function itself or a copy of the provided function object is returned: see the example below, in which an extra argument list is added to the `for_each()` call, which argument is eventually also passed to the function given to `for_each()`. Within `for_each()` the return value of the function that is passed to it is ignored.

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <cctype>

void lowerCase(char &c)                     // 'c' *is* modified
{
    c = static_cast<char>(tolower(c));
}
                                            // 'str' is *not* modified
void capitalizedOutput(std::string const &str)
{
    char    *tmp = strcpy(new char[str.size() + 1], str.c_str());

    std::for_each(tmp + 1, tmp + str.size(), lowerCase);

    tmp[0] = toupper(*tmp);
    std::cout << tmp << " ";
    delete tmp;
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "BRAVO", "charley", "DELTA",  "echo",
            "FOXTROT", "golf", "HOTEL"
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    for_each(sarr, last, capitalizedOutput)("that's all, folks");
    cout << endl;

    return 0;
}
/*
    Generated output:

    Alpha Bravo Charley Delta Echo Foxtrot Golf Hotel That's all, folks
*/
```

- Here is another example using a function object:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <cctype>

void lowerCase(char &c)
```

```
{
    c = tolower(c);
}

class Show
{
    int d_count;

    public:
        Show()
        :
            d_count(0)
        {}

        void operator()(std::string &str)
        {
            std::for_each(str.begin(), str.end(), lowerCase);
            str[0] = toupper(str[0]);   // here assuming str.length()
            std::cout << ++d_count << " " << str << "; ";
        }

        int count() const
        {
            return d_count;
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "BRAVO", "charley", "DELTA", "echo",
            "FOXTROT", "golf", "HOTEL"
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    cout << for_each(sarr, last, Show()).count() << endl;

    return 0;
}
/*
    Generated output (all on a single line):

    1 Alpha; 2 Bravo; 3 Charley; 4 Delta; 5 Echo; 6 Foxtrot;
                                                  7 Golf; 8 Hotel; 8
*/
```

The example also shows that the `for_each` algorithm may be used with functions defining `const` and non-`const` parameters. Also, see section 19.1.63 for differences between the `for_each()` and `transform()` generic algorithms.

The `for_each()` algorithm cannot directly be used (i.e., by passing `*this` as the function object argument) inside a member function to modify its own object as the `for_each()` algorithm first creates its own copy of the passed function object. A *wrapper class* whose constructor accepts a pointer or reference to the current object and possibly to one of its member functions solves this problem. In section 23.8 the construction of such wrapper classes is described.

19.1.18 generate()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void generate(ForwardIterator first, ForwardIterator last, Generator generator);`
- Description:
  - All elements implied by the iterator range `[first, last)` are initialized by the return value of `generator`, which can be a function or function object. `Generator::operator()()` does not receive any arguments. The example uses a well-known fact from algebra: in order to obtain the square of `n + 1`, add `1 + 2 * n` to `n * n`.
- Example:

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>

class NaturalSquares
{
    size_t d_newsqr;
    size_t d_last;

    public:
        NaturalSquares(): d_newsqr(0), d_last(0)
        {}
        size_t operator()()
        {                   // using: (a + 1)^2 == a^2 + 2*a + 1
            return d_newsqr += (d_last++ << 1) + 1;
        }
};

using namespace std;

int main()
{
    vector<size_t>    uv(10);

    generate(uv.begin(), uv.end(), NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    1 4 9 16 25 36 49 64 81 100
*/
```

19.1.19 generate_n()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void generate_n(ForwardIterator first, Size n, Generator generator);`
- Description:
  - `n` elements starting at the element pointed to by iterator `first` are initialized by the return value of `generator`, which can be a function or function object.
- Example:

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>

class NaturalSquares
{
    size_t d_newsqr;
    size_t d_last;

    public:
        NaturalSquares(): d_newsqr(0), d_last(0)
        {}
        size_t operator()()
        {                           // using: (a + 1)^2 == a^2 + 2*a + 1
            return d_newsqr += (d_last++ << 1) + 1;
        }
};

using namespace std;

int main()
{
    vector<size_t>    uv(10);

    generate_n(uv.begin(), 5, NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    1 4 9 16 25 0 0 0 0 0
*/
```

19.1.20 includes()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `bool includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);`
  - `bool includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);`
- Description:
  - The first prototype: both sequences of elements implied by the ranges `[first1, last1)` and `[first2, last2)` should have been sorted using the `operator<()` of the data type to which the iterators point. The function returns `true` if every element in the second sequence `[first2, second2)` is contained in the first sequence `[first1, second1)` (the second range is a subset of the first range).
  - The second prototype: both sequences of elements implied by the ranges `[first1, last1)` and `[first2, last2)` should have been sorted using the `comp` function object. The function returns `true` if every element in the second sequence `[first2, second2)` is contained in the first seqence `[first1, second1)` (the second range is a subset of the first range).
- Example:

```
#include <algorithm>
#include <string>
#include <iostream>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return !strcasecmp(first.c_str(), second.c_str());
        }
};

using namespace std;

int main()
{
    string first1[] =
        {
            "alpha", "bravo", "charley", "delta",  "echo",
            "foxtrot", "golf", "hotel"
        };
    string first2[] =
        {
            "Alpha", "bravo", "Charley", "delta",  "Echo",
            "foxtrot", "Golf", "hotel"
        };
    string second[] =
        {
            "charley", "foxtrot", "hotel"
        };
    size_t n = sizeof(first1) / sizeof(string);

    cout << "The elements of 'second' are " <<
        (includes(first1, first1 + n, second, second + 3) ? "" : "not")
        << " contained in the first sequence:\n"
           "second is a subset of first1\n";
```

```
    cout << "The elements of 'first1' are " <<
        (includes(second, second + 3, first1, first1 + n) ? "" : "not")
        << " contained in the second sequence\n";

    cout << "The elements of 'second' are " <<
        (includes(first2, first2 + n, second, second + 3) ? "" : "not")
        << " contained in the first2 sequence\n";

    cout << "Using case-insensitive comparison,\n"
        "the elements of 'second' are "
        <<
        (includes(first2, first2 + n, second, second + 3, CaseString()) ?
            "" : "not")
        << " contained in the first2 sequence\n";

    return 0;
}
/*
    Generated output:

    The elements of 'second' are  contained in the first sequence:
    second is a subset of first1
    The elements of 'first1' are not contained in the second sequence
    The elements of 'second' are not contained in the first2 sequence
    Using case-insensitive comparison,
    the elements of 'second' are  contained in the first2 sequence
*/
```

19.1.21 inner_product()

- Header file:

```
#include <numeric>
```

- Function prototypes:
  - `Type inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type init);`
  - `Type inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type init, BinaryOperator1 op1, BinaryOperator2 op2);`
- Description:
  - The first prototype: the sum of all pairwise products of the elements implied by the range `[first1, last1)` and the same number of elements starting at the element pointed to by `first2` are added to `init`, and this sum is returned. The function uses the `operator+()` and `operator*()` of the data type to which the iterators point.
  - The second prototype: binary operator `op1` instead of the default addition operator, and binary operator `op2` instead of the default multiplication operator are applied to all pairwise elements implied by the range `[first1, last1)` and the same number of elements starting at the element pointed to by `first2`. The final result is returned.
- Example:

```
#include <numeric>
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>
```

```
class Cat
{
    std::string  d_sep;
    public:
        Cat(std::string const &sep)
        :
            d_sep(sep)
        {}
        std::string operator()
            (std::string const &s1, std::string const &s2) const
        {
            return s1 + d_sep + s2;
        }
};

using namespace std;

int main()
{
    size_t ia1[] = {1, 2, 3, 4, 5, 6, 7};
    size_t ia2[] = {7, 6, 5, 4, 3, 2, 1};
    size_t init = 0;

    cout << "The sum of all squares in ";
    copy(ia1, ia1 + 7, ostream_iterator<size_t>(cout, " "));
    cout << "is " <<
        inner_product(ia1, ia1 + 7, ia1, init) << endl;

    cout << "The sum of all cross-products in ";
    copy(ia1, ia1 + 7, ostream_iterator<size_t>(cout, " "));
    cout << " and ";
    copy(ia2, ia2 + 7, ostream_iterator<size_t>(cout, " "));
    cout << "is " <<
        inner_product(ia1, ia1 + 7, ia2, init) << endl;

    string names1[] = {"Frank", "Karel", "Piet"};
    string names2[] = {"Brokken", "Kubat", "Plomp"};

    cout << "A list of all combined names in ";
    copy(names1, names1 + 3, ostream_iterator<string>(cout, " "));
    cout << "and\n";
    copy(names2, names2 + 3, ostream_iterator<string>(cout, " "));
    cout << "is:" <<
        inner_product(names1, names1 + 3, names2, string("\t"),
            Cat("\n\t"), Cat(" ")) <<
        endl;

    return 0;
}
/*
    Generated output:

    The sum of all squares in 1 2 3 4 5 6 7 is 140
    The sum of all cross-products in 1 2 3 4 5 6 7  and 7 6 5 4 3 2 1 is 84
    A list of all combined names in Frank Karel Piet and
    Brokken Kubat Plomp is:
            Frank Brokken
            Karel Kubat
```

```
                Piet Plomp
    */
```

19.1.22 inplace_merge()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last);`
  - `void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last, Compare comp);`
- Description:
  - The first prototype: the two (sorted) ranges `[first, middle)` and `[middle, last)` are merged, keeping a sorted list (using the `operator<()` of the data type to which the iterators point). The final series is stored in the range `[first, last)`.
  - The second prototype: the two (sorted) ranges `[first, middle)` and `[middle, last)` are merged, keeping a sorted list (using the boolean result of the binary comparison operator `comp`). The final series is stored in the range `[first, last)`.
- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string range[] =
        {
            "alpha", "charley", "echo", "golf",
            "bravo", "delta", "foxtrot",
        };

    inplace_merge(range, range + 4, range + 7);
    copy(range, range + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    string range2[] =
        {
            "ALFA", "CHARLEY", "DELTA", "foxtrot", "hotel",
```

```
            "bravo", "ECHO", "GOLF"
        };

    inplace_merge(range2, range2 + 5, range2 + 8, CaseString());
    copy(range2, range2 + 8, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    alpha bravo charley delta echo foxtrot golf
    ALFA bravo CHARLEY DELTA ECHO foxtrot GOLF hotel
*/
```

19.1.23 iter_swap()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void iter_swap(ForwardIterator1 iter1, ForwardIterator2 iter2);`
- Description:
  - The elements pointed to by `iter1` and `iter2` are swapped.
- Example:

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (size_t idx = 0; idx < n; ++idx)
        iter_swap(first + idx, second + idx);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;
```

```
    return 0;
}
/*
    Generated output:

    Before:
    alpha bravo charley
    echo foxtrot golf
    After:
    echo foxtrot golf
    alpha bravo charley
*/
```

19.1.24 lexicographical_compare()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);`
  - `bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);`
- Description:
  - The first prototype: the corresponding pairs of elements in the ranges pointed to by `[first1, last1)` and `[first2, last2)` are compared. The function returns `true`
    - at the first element in the first range which is less than the corresponding element in the second range (using `operator<()` of the underlying data type),
    - if `last1` is reached, but `last2` isn't reached yet.

    False is returned in the other cases, which indicates that the first sequence is not lexicographically less than the second sequence. So, `false` is returned:
    - at the first element in the first range which is greater than the corresponding element in the second range (using `operator<()` of the data type to which the iterators point, reversing the operands),
    - if `last2` is reached, but `last1` isn't reached yet,
    - if `last1` and `last2` are reached.
  - The second prototype: with this function the binary comparison operation as defined by `comp` is used instead of `operator<()` of the data type to which the iterators point.
- Example:

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
```

```
        }
};

using namespace std;

int main()
{
    string word1 = "hello";
    string word2 = "help";

    cout << word1 << " is " <<
        (
            lexicographical_compare(word1.begin(), word1.end(),
                                    word2.begin(), word2.end()) ?
                "before "
            :
                "beyond or at "
        ) <<
        word2 << " in the alphabet\n";

    cout << word1 << " is " <<
        (
            lexicographical_compare(word1.begin(), word1.end(),
                                    word1.begin(), word1.end()) ?
                "before "
            :
                "beyond or at "
        ) <<
        word1 << " in the alphabet\n";

    cout << word2 << " is " <<
        (
            lexicographical_compare(word2.begin(), word2.end(),
                                    word1.begin(), word1.end()) ?
                "before "
            :
                "beyond or at "
        ) <<
        word1 << " in the alphabet\n";

    string one[] = {"alpha", "bravo", "charley"};
    string two[] = {"ALPHA", "BRAVO", "DELTA"};

    copy(one, one + 3, ostream_iterator<string>(cout, " "));
    cout << " is ordered " <<
        (
            lexicographical_compare(one, one + 3,
                                    two, two + 3, CaseString()) ?
                "before "
            :
                "beyond or at "
        );
    copy(two, two + 3, ostream_iterator<string>(cout, " "));
    cout << endl <<
        "using case-insensitive comparisons.\n";

    return 0;
}
/*
```

```
    Generated output:

    hello is before help in the alphabet
    hello is beyond or at hello in the alphabet
    help is beyond or at hello in the alphabet
    alpha bravo charley  is ordered before ALPHA BRAVO DELTA
    using case-insensitive comparisons.
*/
```

19.1.25 lower_bound()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const Type &value);`
  - `ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const Type &value, Compare comp);`
- Description:
  - The first prototype: the sorted elements indicated by the iterator range `[first, last)` are searched for the first element that is not less than (i.e., greater than or equal to) `value`. The returned iterator marks the location in the sequence where `value` can be inserted without breaking the sorted order of the elements. The `operator<()` of the data type to which the iterators point is used. If no such element is found, `last` is returned.
  - The second prototype: the elements indicated by the iterator range `[first, last)` must have been sorted using the `comp` function (-object). Each element in the range is compared to `value` using the `comp` function. An iterator to the first element for which the binary predicate `comp`, applied to the elements of the range and `value`, returns `false` is returned. If no such element is found, `last` is returned.
- Example:

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
using namespace std;

int main()
{
    int     ia[] = {10, 20, 30};

    cout << "Sequence: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
            *lower_bound(ia, ia + 3, 15) << endl;
    cout << "35 can be inserted after " <<
            (lower_bound(ia, ia + 3, 35) == ia + 3 ?
                                "the last element" : "???") << endl;

    iter_swap(ia, ia + 2);
```

```
    cout << "Sequence: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
            *lower_bound(ia, ia + 3, 15, greater<int>()) << endl;
    cout << "35 can be inserted before " <<
            (lower_bound(ia, ia + 3, 35, greater<int>()) == ia ?
                                "the first element " : "???") << endl;

    return 0;
}
/*
    Generated output:

    Sequence: 10 20 30
    15 can be inserted before 20
    35 can be inserted after the last element
    Sequence: 30 20 10
    15 can be inserted before 10
    35 can be inserted before the first element
*/
```

19.1.26 max()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `Type const &max(Type const &one, Type const &two);`
  - `Type const &max(Type const &one, Type const &two, Comparator comp);`
- Description:
  - The first prototype: the larger of the two elements `one` and `two` is returned using the `operator>()` of the data type to which the iterators point.
  - The second prototype: `one` is returned if the binary predicate `comp(one, two)` returns `true`, otherwise `two` is returned.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(second.c_str(), first.c_str()) > 0;
        }
};

using namespace std;
```

```
int main()
{
    cout << "Word '" << max(string("first"), string("second")) <<
                                "' is lexicographically last\n";

    cout << "Word '" << max(string("first"), string("SECOND")) <<
                                "' is lexicographically last\n";

    cout << "Word '" << max(string("first"), string("SECOND"),
                       CaseString()) << "' is lexicographically last\n";

    return 0;
}
/*
    Generated output:

    Word 'second' is lexicographically last
    Word 'first' is lexicographically last
    Word 'SECOND' is lexicographically last
*/
```

19.1.27 max_element()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator max_element(ForwardIterator first, ForwardIterator last);`
  - `ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Comparator comp);`
- Description:
  - The first prototype: an iterator pointing to the largest element in the range implied by `[first, last)` is returned. The `operator<()` of the data type to which the iterators point is used.
  - The second prototype: rather than using `operator<()`, the binary predicate `comp` is used to make the comparisons between the elements implied by the iterator range `[first, last)`. The element for which `comp` returns most often `true`, compared with other elements, is returned.
- Example:

```
#include <algorithm>
#include <iostream>

class AbsValue
{
    public:
        bool operator()(int first, int second) const
        {
            return abs(first) < abs(second);
        }
};

using namespace std;
```

```
int main()
{
    int     ia[] = {-4, 7, -2, 10, -12};

    cout << "The max. int value is " << *max_element(ia, ia + 5) << endl;
    cout << "The max. absolute int value is " <<
            *max_element(ia, ia + 5, AbsValue()) << endl;

    return 0;
}
/*
    Generated output:

    The max. int value is 10
    The max. absolute int value is -12
*/
```

19.1.28 merge()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: the two (sorted) ranges `[first1, last1)` and `[first2, last2)` are merged, keeping a sorted list (using the `operator<()` of the data type to which the iterators point). The final series is stored in the range starting at `result` and ending just before the `OutputIterator` returned by the function.
  - The first prototype: the two (sorted) ranges `[first1, last1)` and `[first2, last2)` are merged, keeping a sorted list (using the boolean result of the binary comparison operator `comp`). The final series is stored in the range starting at `result` and ending just before the `OutputIterator` returned by the function.
- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};
```

```
using namespace std;

int main()
{
    string range1[] =
        {                                       // 5 elements
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        };
    string range2[] =
        {                                       // 4 elements
            "echo", "delta", "golf", "romeo"
        };
    string result[5 + 4];


    copy(result,
        merge(range1, range1 + 5, range2, range2 + 4, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string range3[] =
        {
            "ALPHA", "bravo", "foxtrot", "HOTEL", "ZULU"
        };
    string range4[] =
        {
            "delta", "ECHO", "GOLF", "romeo"
        };

    copy(result,
        merge(range3, range3 + 5, range4, range4 + 4, result,
                                                    CaseString()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    alpha bravo echo delta foxtrot golf hotel romeo zulu
    ALPHA bravo delta ECHO foxtrot GOLF HOTEL romeo ZULU
*/
```

19.1.29 min()

- Header file:

  ```
  #include <algorithm>
  ```

- Function prototypes:
  - `Type const &min(Type const &one, Type const &two);`
  - `Type const &min(Type const &one, Type const &two, Comparator comp);`
- Description:
  - The first prototype: the smaller of the two elements `one` and `two` is returned using the `operator<()` of the data type to which the iterators point.

  - The second prototype: `one` is returned if the binary predicate `comp(one, two)` returns `false`, otherwise `two` is returned.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(second.c_str(), first.c_str()) > 0;
        }
};

using namespace std;

int main()
{
    cout << "Word '" << min(string("first"), string("second")) <<
                                "' is lexicographically first\n";

    cout << "Word '" << min(string("first"), string("SECOND")) <<
                                "' is lexicographically first\n";

    cout << "Word '" << min(string("first"), string("SECOND"),
                        CaseString()) << "' is lexicographically first\n";

    return 0;
}
/*
    Generated output:

    Word 'first' is lexicographically first
    Word 'SECOND' is lexicographically first
    Word 'first' is lexicographically first
*/
```

19.1.30 min_element()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator min_element(ForwardIterator first, ForwardIterator last);`
  - `ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Comparator comp);`

- Description:
  - The first prototype: an iterator pointing to the smallest element in the range implied by `[first, last)` is returned using `operator<()` of the data type to which the iterators point.

- The second prototype: rather than using `operator<()`, the binary predicate `comp` is used to make the comparisons between the elements implied by the iterator range `[first, last)`. The element for which `comp` returns `false` most often is returned.

- Example:

```
#include <algorithm>
#include <iostream>

class AbsValue
{
    public:
        bool operator()(int first, int second) const
        {
            return abs(first) < abs(second);
        }
};

using namespace std;

int main()
{
    int     ia[] = {-4, 7, -2, 10, -12};

    cout << "The minimum int value is " << *min_element(ia, ia + 5) <<
            endl;
    cout << "The minimum absolute int value is " <<
            *min_element(ia, ia + 5, AbsValue()) << endl;

    return 0;
}
/*
    Generated output:

    The minimum int value is -12
    The minimum absolute int value is -2
*/
```

19.1.31 mismatch()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);`
  - `pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Compare comp);`

- Description:
  - The first prototype: the two sequences of elements starting at `first1` and `first2` are compared using the equality operator of the data type to which the iterators point. Comparison stops if the compared elements differ (i.e., `operator==()` returns false) or `last1` is reached. A `pair` containing iterators pointing to the final positions is returned. The second sequence may contain more elements than the first sequence. The behavior of the algorithm is undefined if the second sequence contains fewer elements than the first sequence.

- The second prototype: the two sequences of elements starting at `first1` and `first2` are compared using the binary comparison operation as defined by `comp`, instead of `operator==()`. Comparison stops if the `comp` function returns `false` or `last1` is reached. A `pair` containing iterators pointing to the final positions is returned. The second sequence may contain more elements than the first sequence. The behavior of the algorithm is undefined if the second sequence contains fewer elements than the first sequence.

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <utility>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) == 0;
        }
};

using namespace std;

int main()
{
    string range1[] =
        {
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        };
    string range2[] =
        {
            "alpha", "bravo", "foxtrot", "Hotel", "zulu"
        };
    pair<string *, string *> pss = mismatch(range1, range1 + 5, range2);

    cout << "The elements " << *pss.first << " and " << *pss.second <<
            " at offset " << (pss.first - range1) << " differ\n";
    if
    (
        mismatch(range1, range1 + 5, range2, CaseString()).first
        ==
        range1 + 5
    )
        cout << "When compared case-insensitively they match\n";

    return 0;
}
/*
    Generated output:

    The elements hotel and Hotel at offset 3 differ
    When compared case-insensitively they match
*/
```

19.1.32 next_permutation()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);`
  - `bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, Comp comp);`
- Description:
  - The first prototype: the next permutation, given the sequence of elements in the range `[first, last)`, is determined. For example, if the elements `1, 2` and `3` are the range for which `next_permutation()` is called, then subsequent calls of `next_permutation()` reorders the following series:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

    This example shows that the elements are reordered such that each new permutation represents the next bigger value (132 is bigger than 123, 213 is bigger than 132, etc.) using `operator<()` of the data type to which the iterators point. The value `true` is returned if a reordering took place, the value `false` is returned if no reordering took place, which is the case if the sequence represents the last (biggest) value. In that case, the sequence is also sorted using `operator<()`.
  - The second prototype: the next permutation given the sequence of elements in the range `[first, last)` is determined. The elements in the range are reordered. The value `true` is returned if a reordering took place, the value `false` is returned if no reordering took place, which is the case if the resulting sequence would haven been ordered using the binary predicate `comp` to compare elements.
  - Example:

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string saints[] = {"Oh", "when", "the", "saints"};
```

```
    cout << "All permutations of 'Oh when the saints':\n";

    cout << "Sequences:\n";
    do
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << endl;
    }
    while (next_permutation(saints, saints + 4, CaseString()));


    cout << "After first sorting the sequence:\n";

    sort(saints, saints + 4, CaseString());

    cout << "Sequences:\n";
    do
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << endl;
    }
    while (next_permutation(saints, saints + 4, CaseString()));

    return 0;
}
/*
    Generated output (only partially given):

    All permutations of 'Oh when the saints':
    Sequences:
    Oh when the saints
    saints Oh the when
    saints Oh when the
    saints the Oh when
    ...
    After first sorting the sequence:
    Sequences:
    Oh saints the when
    Oh saints when the
    Oh the saints when
    Oh the when saints
    ...
*/
```

19.1.33 nth_element()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);`
  - `void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);`
- Description:

- The first prototype: all elements in the range [first, last) are sorted relative to the element pointed to by nth: all elements in the range [left, nth) are smaller than the element pointed to by nth, and alle elements in the range [nth + 1, last) are greater than the element pointed to by nth. The two subsets themselves are not sorted. The operator<() of the data type to which the iterators point is used to compare the elements.
- The second prototype: all elements in the range [first, last) are sorted relative to the element pointed to by nth: all elements in the range [left, nth) are smaller than the element pointed to by nth, and alle elements in the range [nth + 1, last) are greater than the element pointed to by nth. The two subsets themselves are not sorted. The comp function object is used to compare the elements.

- Example:

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    nth_element(ia, ia + 3, ia + 10);

    cout << "sorting with respect to " << ia[3] << endl;
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    nth_element(ia, ia + 5, ia + 10, greater<int>());

    cout << "sorting with respect to " << ia[5] << endl;
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    sorting with respect to 4
    1 2 3 4 9 7 5 6 8 10
    sorting with respect to 5
    10 8 7 9 6 5 3 4 2 1
*/
```

19.1.34 partial_sort()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);`
  - `void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp);`

- Description:
  - The first prototype: the `middle - first` smallest elements are sorted and stored in the `[first, middle)` using the `operator<()` of the data type to which the iterators point. The remaining elements of the series remain unsorted, and are stored in `[middle, last)`.
  - The second prototype: the `middle - first` smallest elements (according to the provided binary predicate `comp`) are sorted and stored in the `[first, middle)`. The remaining elements of the series remain unsorted.
- Example:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    partial_sort(ia, ia + 3, ia + 10);

    cout << "find the 3 smallest elements:\n";
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "find the 5 biggest elements:\n";
    partial_sort(ia, ia + 5, ia + 10, greater<int>());
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    find the 3 smallest elements:
    1 2 3 7 9 5 4 6 8 10
    find the 5 biggest elements:
    10 9 8 7 6 1 2 3 4 5
*/
```

19.1.35 partial_sort_copy()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator dest_first, RandomAccessIterator dest_last);`
  - `void partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator dest_first, RandomAccessIterator dest_last, Compare comp);`

- Description:
  - The first prototype: the smallest elements in the range [first, last) are copied to the range [dest_first, dest_last), using the operator<() of the data type to which the iterators point. Only the number of elements in the smaller range are copied to the second range.
  - The second prototype: the elements in the range [first, last) are are sorted by the binary predicate comp. The elements for which the predicate returns most often true are copied to the range [dest_first, dest_last). Only the number of elements in the smaller range are copied to the second range.
- Example:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 10, 3, 8, 5, 6, 7, 4, 9, 2};
    int ia2[6];

    partial_sort_copy(ia, ia + 10, ia2, ia2 + 6);

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;
    cout << "the 6 smallest elements: ";
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "the 4 smallest elements to a larger range:\n";
    partial_sort_copy(ia, ia + 4, ia2, ia2 + 6);
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "the 4 biggest elements to a larger range:\n";
    partial_sort_copy(ia, ia + 4, ia2, ia2 + 6, greater<int>());
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    1 10 3 8 5 6 7 4 9 2
    the 6 smallest elements: 1 2 3 4 5 6
    the 4 smallest elements to a larger range:
    1 3 8 10 5 6
    the 4 biggest elements to a larger range:
    10 8 3 1 5 6
*/
```

19.1.36 partial_sum()

- Header file:

```
#include <numeric>
```

- Function prototypes:
  - `OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result);`
  - `OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);`
- Description:
  - The first prototype: each element in the range `[result, <returned OutputIterator>)` receives a value which is obtained by adding the elements in the corresponding range of the range `[first, last)`. The first element in the resulting range will be equal to the element pointed to by `first`.
  - The second prototype: the value of each element in the range `[result, <returned OutputIterator>)` is obtained by applying the binary operator `op` to the previous element in the resulting range and the corresponding element in the range `[first, last)`. The first element in the resulting range will be equal to the element pointed to by `first`.
- Example:

```
#include <numeric>
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 5};
    int ia2[5];

    copy(ia2,
        partial_sum(ia, ia + 5, ia2),
        ostream_iterator<int>(cout, " "));
    cout << endl;

    copy(ia2,
        partial_sum(ia, ia + 5, ia2, multiplies<int>()),
        ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    1 3 6 10 15
    1 2 6 24 120
*/
```

19.1.37 partition()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `BidirectionalIterator partition(BidirectionalIterator first, BidirectionalIterator last, UnaryPredicate pred);`
- Description:
  - All elements in the range `[first, last)` for which the unary predicate `pred` evaluates as `true` are placed before the elements which evaluate as `false`. The return value points just beyond the last element in the partitioned range for which `pred` evaluates as `true`.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class LessThan
{
    int d_x;
    public:
        LessThan(int x)
        :
            d_x(x)
        {}
        bool operator()(int value)
        {
            return value <= d_x;
        }
};

using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4};
    int *split;

    split = partition(ia, ia + 10, LessThan(ia[9]));
    cout << "Last element <= 4 is ia[" << split - ia - 1 << "]\n";

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Last element <= 4 is ia[3]
    1 3 4 2 9 10 7 8 6 5
*/
```

19.1.38 prev_permutation()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);`
  - `bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last, Comp comp);`
- Description:
  - The first prototype: the previous permutation given the sequence of elements in the range `[first, last)` is determined. The elements in the range are reordered such that the first ordering is obtained representing a 'smaller' value (see `next_permutation()` (section 19.1.32) for an example involving the opposite ordering). The value `true` is returned if a reordering took place, the value `false` is returned if no reordering took place, which is the case if the provided sequence was already ordered, according to the `operator<()` of the data type to which the iterators point.
  - The second prototype: the previous permutation given the sequence of elements in the range `[first, last)` is determined. The elements in the range are reordered. The value `true` is returned if a reordering took place, the value `false` is returned if no reordering took place, which is the case if the original sequence was already ordered, using the binary predicate `comp` to compare two elements.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string  saints[] = {"Oh", "when", "the", "saints"};

    cout << "All previous permutations of 'Oh when the saints':\n";

    cout << "Sequences:\n";
    do
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << endl;
    }
    while (prev_permutation(saints, saints + 4, CaseString()));

    cout << "After first sorting the sequence:\n";
    sort(saints, saints + 4, CaseString());

    cout << "Sequences:\n";
    while (prev_permutation(saints, saints + 4, CaseString()))
```

```
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << endl;
    }
    cout << "No (more) previous permutations\n";

    return 0;
}
/*
    Generated output:

    All previous permutations of 'Oh when the saints':
    Sequences:
    Oh when the saints
    Oh when saints the
    Oh the when saints
    Oh the saints when
    Oh saints when the
    Oh saints the when
    After first sorting the sequence:
    Sequences:
    No (more) previous permutations
*/
```

19.1.39 random_shuffle()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);`
  - `void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator rand);`
- Description:
  - The first prototype: the elements in the range `[first, last)` are randomly reordered.
  - The second prototype: The elements in the range `[first, last)` are randomly reordered using the `rand` random number generator, which should return an `int` in the range `[0, remaining)`, where `remaining` is passed as argument to the `operator()()` of the `rand` function object. Alternatively, the random number generator may be a function expecting an `int remaining` parameter and returning an `int` randomvalue in the range `[0, remaining)`. Note that when a function object is used, it cannot be an anonymous object. The function in the example uses a procedure outlined in *Press et al.* (1992) **Numerical Recipes in C: The Art of Scientific Computing** (New York: Cambridge University Press, (2nd ed., p. 277)).
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <time.h>
#include <iterator>

int randomValue(int remaining)
{
```

```
    return static_cast<int>
            ( ((0.0 + remaining) * rand()) / (RAND_MAX + 1.0) );
}

class RandomGenerator
{
    public:
        RandomGenerator()
        {
            srand(time(0));
        }
        int operator()(int remaining) const
        {
            return randomValue(remaining);
        }
};


void show(std::string *begin, std::string *end)
{
    std::copy(begin, end,
                    std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << std::endl << std::endl;
}

using namespace std;

int main()
{
    string words[] =
                { "kilo", "lima", "mike", "november", "oscar", "papa"};
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Using Default Shuffle:\n";
    random_shuffle(words, words + size);
    show(words, words + size);

    cout << "Using RandomGenerator:\n";
    RandomGenerator rg;
    random_shuffle(words, words + size, rg);
    show(words, words + size);

    srand(time(0) << 1);
    cout << "Using the randomValue() function:\n";
    random_shuffle(words, words + size, randomValue);
    show(words, words + size);

    return 0;
}
/*
    Generated output (for example):

    Using Default Shuffle:
    lima oscar mike november papa kilo

    Using RandomGenerator:
    kilo lima papa oscar mike november

    Using the randomValue() function:
```

```
    mike papa november kilo oscar lima
*/
```

19.1.40 remove()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `ForwardIterator remove(ForwardIterator first, ForwardIterator last, Type const &value);`
- Description:
  - The elements in the range pointed to by `[first, last)` are reordered in such a way that all values unequal to `value` are placed at the beginning of the range. The returned forward iterator points to the first element that can be removed after reordering. The range `[returnvalue, last)` is called the *leftover* of the algorithm. Note that the leftover may contain elements different from `value`, but these elements can be removed safely, as such elements will also be present in the range `[first, return value)`. Such duplication is the result of the fact that the algorithm *copies*, rather than *moves* elements into new locations. The function uses `operator==()` of the data type to which the iterators point to determine which elements to remove.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "alpha", "alpha", "papa", "quebec" };
    string *removed;
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Removing all \"alpha\"s:\n";
    removed = remove(words, words + size, "alpha");
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Leftover elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
    Trailing elements are:
    oscar alpha alpha papa quebec
*/
```

19.1.41 remove_copy()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result, Type const &value);`
- Description:
  - The elements in the range pointed to by `[first, last)` not matching `value` are copied to the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The range `[first, last)` is not modified. The function uses `operator==()` of the data type to which the iterators point to determine which elements not to copy.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining
            [
                size -
                count_if
                (
                    words, words + size,
                    bind2nd(equal_to<string>(), string("alpha"))
                )
            ];
    string *returnvalue =
            remove_copy(words, words + size, remaining, "alpha");

    cout << "Removing all \"alpha\"s:\n";
    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
*/
```

19.1.42 remove_copy_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `OutputIterator remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, UnaryPredicate pred);`
- Description:
  - The elements in the range pointed to by `[first, last)` for which the unary predicate `pred` returns `true` are copied to the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The range `[first, last)` is not modified.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining[
                        size -
                        count_if
                        (
                            words, words + size,
                            bind2nd(equal_to<string>(), "alpha")
                        )
                    ];
    string *returnvalue =
                remove_copy_if
                (
                    words, words + size, remaining,
                    bind2nd(equal_to<string>(), "alpha")
                );

    cout << "Removing all \"alpha\"s:\n";
    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
*/
```

19.1.43 remove_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred);`
- Description:
  - The elements in the range pointed to by `[first, last)` are reordered in such a way that all values for which the unary predicate `pred` evaluates as `false` are placed at the beginning of the range. The returned forward iterator points to the first element, after reordering, for which `pred` returns `true`. The range `[returnvalue, last)` is called the *leftover* of the algorithm. The leftover may contain elements for which the predicate `pred` returns `false`, but these can safely be removed, as such elements will also be present in the range `[first, returnvalue)`. Such duplication is the result of the fact that the algorithm *copies*, rather than *moves* elements into new locations.
- Example:

```
#include <functional>
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Removing all \"alpha\"s:\n";

    string *removed = remove_if(words, words + size,
                bind2nd(equal_to<string>(), string("alpha")));

    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
    Trailing elements are:
    oscar alpha alpha papa quebec
*/
```

19.1.44 replace()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `ForwardIterator replace(ForwardIterator first, ForwardIterator last, Type const &oldvalue, Type const &newvalue);`
- Description:
  - All elements equal to `oldvalue` in the range pointed to by `[first, last)` are replaced by a copy of `newvalue`. The algorithm uses `operator==()` of the data type to which the iterators point.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    replace(words, words + size, string("alpha"), string("ALPHA"));
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/
```

19.1.45 replace_copy()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `OutputIterator replace_copy(InputIterator first, InputIterator last, OutputIterator result, Type const &oldvalue, Type const &newvalue);`
- Description:
  - All elements equal to `oldvalue` in the range pointed to by `[first, last)` are replaced by a copy of `newvalue` in a new range `[result, returnvalue)`, where `returnvalue` is the return value of the function. The algorithm uses `operator==()` of the data type to which the iterators point.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining[size];

    copy
    (
        remaining,
        replace_copy(words, words + size, remaining, string("alpha"),
                                                     string("ALPHA")),
        ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/
```

19.1.46 replace_copy_if()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator replace_copy_if(ForwardIterator first, ForwardIterator last, OutputIterator result, UnaryPredicate pred, Type const &value);`
- Description:
  - The elements in the range pointed to by `[first, last)` are copied to the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The elements for which the unary predicate `pred` returns `true` are replaced by `newvalue`. The range `[first, last)` is not modified.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;
```

```
int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november",
          "alpha", "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string result[size];

    replace_copy_if(words, words + size, result,
                    bind1st(greater<string>(), string("mike")),
                    string("ALPHA"));
    copy (result, result + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output (all on one line):

    ALPHA ALPHA ALPHA mike ALPHA november ALPHA oscar ALPHA ALPHA
                                                        papa quebec
*/
```

19.1.47 replace_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `ForwardIterator replace_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred, Type const &value);`
- Description:
  - The elements in the range pointed to by `[first, last)` for which the unary predicate `pred` evaluates as `true` are replaced by `newvalue`.

Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    replace_if(words, words + size,
               bind1st(equal_to<string>(), string("alpha")),
               string("ALPHA"));
```

```
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
    return 0;
}
/*
    generated output:

    kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/
```

19.1.48 reverse()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void reverse(BidirectionalIterator first, BidirectionalIterator last);`
- Description:
  - The elements in the range pointed to by `[first, last)` are reversed.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;

    while (getline(cin, line))
    {
        reverse(line.begin(), line.end());
        cout << line << endl;
    }

    return 0;
}
```

19.1.49 reverse_copy()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result);`
- Description:
  - The elements in the range pointed to by `[first, last)` are copied to the range `[result, returnvalue)` in reversed order. The value `returnvalue` is the value that is returned by the function.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;

    while (getline(cin, line))
    {
        size_t    size = line.size();
        char        copy[size + 1];

        cout << "line: " << line << endl <<
                "reversed: ";
        reverse_copy(line.begin(), line.end(), copy);
        copy[size] = 0;     // 0 is not part of the reversed
                            // line !
        cout << copy << endl;
    }
    return 0;
}
```

19.1.50 rotate()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);`
- Description:
  - The elements implied by the range `[first, middle)` are moved to the end of the container, the elements implied by the range `[middle, last)` are moved to the beginning of the container, keeping the order of the elements in the two subsets intact.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    size_t const size = sizeof(words) / sizeof(string);
    size_t const midsize = 6;
```

```
    rotate(words, words + midsize, words + size);

    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    echo foxtrot golf hotel india juliet kilo lima mike november oscar papa
*/
```

19.1.51 rotate_copy()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);`
- Description:
  - The elements implied by the range `[middle, last)` and then the elements implied by the range `[first, middle)` are copied to the destination container having range `[result, returnvalue)`, where `returnvalue` is the iterator returned by the function. The original order of the elements in the two subsets is not altered.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    size_t const size = sizeof(words) / sizeof(string);
    size_t midsize = 6;
    string out[size];

    copy(out,
        rotate_copy(words, words + midsize, words + size, out),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    echo foxtrot golf hotel india juliet kilo lima mike november oscar papa
*/
```

19.1.52 search()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);`
  - `ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);`
- Description:
  - The first prototype: an iterator into the first range `[first1, last1)` is returned where the elements in the range `[first2, last2)` are found using `operator==()` operator of the data type to which the iterators point. If no such location exists, `last1` is returned.
  - The second prototype: an iterator into the first range `[first1, last1)` is returned where the elements in the range `[first2, last2)` are found using the provided binary predicate `pred` to compare the elements in the two ranges. If no such location exists, `last1` is returned.
- Example:

```
#include <algorithm>
#include <iostream>
#include <iterator>

class absInt
{
    public:
        bool operator()(int i1, int i2)
        {
            return abs(i1) == abs(i2);
        }
};

using namespace std;

int main()
{
    int range1[] =  {-2, -4, -6, -8, 2, 4, 6, 8};
    int range2[] =  {6, 8};

    copy
    (
        search(range1, range1 + 8, range2, range2 + 2),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    copy
    (
        search(range1, range1 + 8, range2, range2 + 2, absInt()),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;
```

```
    return 0;
}
/*
    Generated output:

    6 8
    -6 -8 2 4 6 8
*/
```

19.1.53 search_n()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator1 search_n(ForwardIterator1 first1, ForwardIterator1 last1, Size count, Type const &value);`
  - `ForwardIterator1 search_n(ForwardIterator1 first1, ForwardIterator1 last1, Size count, Type const &value, BinaryPredicate pred);`
- Description:
  - The first prototype: an iterator into the first range `[first1, last1)` is returned where `n` consecutive elements having value `value` are found using `operator==()` of the data type to which the iterators point to compare the elements. If no such location exists, `last1` is returned.
  - The second prototype: an iterator into the first range `[first1, last1)` is returned where `n` consecutive elements having value `value` are found using the provided binary predicate `pred` to compare the elements. If no such location exists, `last1` is returned.
- Example:

```
#include <algorithm>
#include <iostream>
#include <iterator>

class absInt
{
    public:
        bool operator()(int i1, int i2)
        {
            return abs(i1) == abs(i2);
        }
};

using namespace std;

int main()
{
    int range1[] =  {-2, -4, -4, -6, -8, 2, 4, 4, 6, 8};
    int range2[] =  {6, 8};

    copy
    (
        search_n(range1, range1 + 8, 2, 4),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
```

```
    );
    cout << endl;

    copy
    (
        search_n(range1, range1 + 8, 2, 4, absInt()),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Generated output:

    4 4
    -4 -4 -6 -8 2 4 4
*/
```

19.1.54 set_difference()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the data type to which the iterators point.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
    public:
        bool operator()(std::string const &left, std::string const &right)
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
```

```
        }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                      "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *returned;

    copy(result,
        set_difference(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};

    copy(result,
        set_difference(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo lima mike november oscar
    kilo lima mike november oscar
*/
```

19.1.55 set_intersection()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are also present in the range `[first2, last2)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the data type to which the iterators point.

  - The second prototype: a sorted sequence of the elements pointed to by the range [first1, last1) that are also present in the range [first2, last2) is returned, starting at result, and ending at the OutputIterator returned by the function. The elements in the two ranges must have been sorted using the comp function object.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
    public:
        bool operator()(std::string const &left, std::string const &right)
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                      "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *returned;

    copy(result,
        set_intersection(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};

    copy(result,
        set_intersection(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    papa quebec
    papa quebec
*/
```

19.1.56 set_symmetric_difference()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator set_symmetric_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_symmetric_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` and those in the range `[first2, last2)` that are not present in the range `[first1, last1)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the data type to which the iterators point.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` and those in the range `[first2, last2)` that are not present in the range `[first1, last1)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
    public:
        bool operator()(std::string const &left, std::string const &right)
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                      "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *returned;

    copy(result,
        set_symmetric_difference(set1, set1 + 7, set2, set2 + 3,
                                                        result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};

    copy(result,
        set_symmetric_difference(set1, set1 + 7, set3, set3 + 3,
```

```
                                                        result,
            CaseLess()),
            ostream_iterator<string>(cout, " "));
        cout << endl;

        return 0;
    }
    /*
        Generated output:

        kilo lima mike november oscar romeo
        kilo lima mike november oscar ROMEO
    */
```

19.1.57 set_union()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: a sorted sequence of the elements that are present in either the range `[first1, last1)` or the range `[first2, last2)` or in both ranges is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the data type to which the iterators point. Note that in the final range each element will appear only once.
  - The second prototype: a sorted sequence of the elements that are present in either the range `[first1, last1)` or the range `[first2, last2)` or in both ranges is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `comp` function object. Note that in the final range each element will appear only once.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
    public:
        bool operator()(std::string const &left, std::string const &right)
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

using namespace std;
```

```
int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                      "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *returned;

    copy(result,
        set_union(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};

    copy(result,
        set_union(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo lima mike november oscar papa quebec romeo
    kilo lima mike november oscar papa quebec ROMEO
*/
```

19.1.58 sort()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void sort(RandomAccessIterator first, RandomAccessIterator last);`
  - `void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: the elements in the range `[first, last)` are sorted in ascending order using `operator<()` of the data type to which the iterators point.
  - The second prototype: the elements in the range `[first, last)` are sorted in ascending order using the `comp` function object to compare the elements. The binary predicate `comp` should return `true` if its first argument should be placed earlier in the sorted sequence than its second argument.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;
```

```
int main()
{
    string words[] = {"november", "kilo", "mike", "lima",
                      "oscar", "quebec", "papa"};

    sort(words, words + 7);
    copy(words, words + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    sort(words, words + 7, greater<string>());
    copy(words, words + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo lima mike november oscar papa quebec
    quebec papa oscar november mike lima kilo
*/
```

19.1.59 stable_partition()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `BidirectionalIterator stable_partition(BidirectionalIterator first, BidirectionalIterator last, UnaryPredicate pred);`
- Description:
  - All elements in the range `[first, last)` for which the unary predicate `pred` evaluates as `true` are placed before the elements which evaluate as `false`. Apart from this reordering, the relative order of all elements for which the predicate evaluates to `false` and the relative order of all elements for which the predicate evaluates to `true` is kept. The return value points just beyond the last element in the partitioned range for which `pred` evaluates as `true`.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int org[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4};
    int ia[10];
    int *split;

    copy(org, org + 10, ia);
```

```
    split = partition(ia, ia + 10, bind2nd(less_equal<int>(), ia[9]));
    cout << "Last element <= 4 is ia[" << split - ia - 1 << "]\n";

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    copy(org, org + 10, ia);
    split = stable_partition(ia, ia + 10,
                             bind2nd(less_equal<int>(), ia[9]));
    cout << "Last element <= 4 is ia[" << split - ia - 1 << "]\n";

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Last element <= 4 is ia[3]
    1 3 4 2 9 10 7 8 6 5
    Last element <= 4 is ia[3]
    1 3 2 4 5 7 9 10 8 6
*/
```

19.1.60 stable_sort()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void stable_sort(RandomAccessIterator first, RandomAccessIterator last);`
  - `void stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: the elements in the range `[first, last)` are stable-sorted in ascending order using `operator<()` of the data type to which the iterators point: the relative order of equal elements is kept.
  - The second prototype: the elements in the range `[first, last)` are stable-sorted in ascending order using the `comp` binary predicate to compare the elements. This predicate should return `true` if its first argument should be placed before its second argument in the sorted set of element.
- Example (annotated below):

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <iterator>

typedef std::pair<std::string, std::string>  pss;   // 1 (see the text)

namespace std
{
```

```
    ostream &operator<<(ostream &out, pss const &p)                 // 2
    {
        return out << "    " << p.first << " " << p.second << endl;
    }
}

class sortby
{
    std::string pss::*d_field;
    public:
        sortby(std::string pss::*field)                             // 3
        :
            d_field(field)
        {}

        bool operator()(pss const &p1, pss const &p2) const         // 4
        {
            return p1.*d_field < p2.*d_field;
        }
};

using namespace std;

int main()
{
    vector<pss> namecity;                                           // 5

    namecity.push_back(pss("Hampson",   "Godalming"));
    namecity.push_back(pss("Moran",     "Eugene"));
    namecity.push_back(pss("Goldberg",  "Eugene"));
    namecity.push_back(pss("Moran",     "Godalming"));
    namecity.push_back(pss("Goldberg",  "Chicago"));
    namecity.push_back(pss("Hampson",   "Eugene"));

    sort(namecity.begin(), namecity.end(), sortby(&pss::first));    // 6

    cout << "sorted by names:\n";
    copy(namecity.begin(), namecity.end(), ostream_iterator<pss>(cout));

                                                                    // 7
    stable_sort(namecity.begin(), namecity.end(), sortby(&pss::second));

    cout << "sorted by names within sorted cities:\n";
    copy(namecity.begin(), namecity.end(), ostream_iterator<pss>(cout));

    return 0;
}
/*
    Generated output:

    sorted by names:
        Goldberg Eugene
        Goldberg Chicago
        Hampson Godalming
        Hampson Eugene
        Moran Eugene
        Moran Godalming
    sorted by names within sorted cities:
        Goldberg Chicago
```

```
            Goldberg Eugene
            Hampson Eugene
            Moran Eugene
            Hampson Godalming
            Moran Godalming
    */
```

Note that the example implements a solution to an often occurring problem: how to sort using multiple hierarchal criteria. The example deserves some additional attention:

1. First, a `typedef` is used to reduce the clutter that occurs from the repeated use of `pair<string, string>`.
2. Next, `operator<<` is overloaded to be able to insert a `pair` into an `ostream` object. This is merely a service function to make life easy. Note, however, that this function is put in the `std` namespace. If this namespace wrapping is omitted, it won't be used, as `ostream`'s `operator<<` operators must be part of the `std` namespace.
3. Then, a class `sortby` is defined, allowing us to construct an anonymous object which receives a pointer to one of the `pair` data members that are used for sorting. In this case, as both members are `string` objects, the constructor can easily be defined: its parameter is a pointer to a `string` member of the class `pair<string, string>`.
4. The `operator()()` member will receive two `pair` references, and it will then use the pointer to its members, stored in the `sortby` object, to compare the appropriate fields of the `pairs`.
5. In `main()`, first some data is stored in a `vector`.
6. Then the first sorting takes place. The least important criterion must be sorted first, and for this a simple `sort()` will suffice. Since we want the names to be sorted within cities, the names represent the least important criterion, so we sort by names: `sortby(&pss::first)`.
7. The next important criterion, the cities, are sorted next. Since the relative ordering of the *names* will not be altered anymore by `stable_sort()`, the ties that are observed when cities are sorted are solved in such a way that the existing relative ordering will not be broken. So, we end up getting Goldberg in Eugene before Hampson in Eugene, before Moran in Eugene. To sort by cities, we use another anonymous `sortby` object: `sortby(&pss::second)`.

19.1.61 swap()

- Header file:

  ```
  #include <algorithm>
  ```

- Function prototype:
  - `void swap(Type &object1, Type &object2);`
- Description:
  - The elements `object1` and `object2` exchange their values.
- Example:

  ```
  #include <algorithm>
  #include <iostream>
  #include <string>
  #include <iterator>
  using namespace std;

  int main()
  ```

```
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (size_t idx = 0; idx < n; ++idx)
        swap(first[idx], second[idx]);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Before:
    alpha bravo charley
    echo foxtrot golf
    After:
    echo foxtrot golf
    alpha bravo charley
*/
```

19.1.62 swap_ranges()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 result);`
- Description:
  - The elements in the range pointed to by `[first1, last1)` are swapped with the elements in the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The two ranges must be disjoint.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
```

```
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    swap_ranges(first, first + n, second);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Before:
    alpha bravo charley
    echo foxtrot golf
    After:
    echo foxtrot golf
    alpha bravo charley
*/
```

19.1.63 transform()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperator op);`
  - `OutputIterator transform(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryOperator op);`
- Description:
  - The first prototype: the unary operator `op` is applied to each of the elements in the range `[first, last)`, and the resulting values are stored in the range starting at `result`. The return value points just beyond the last generated element.
  - The second prototype: the binary operator `op` is applied to each of the elements in the range `[first1, last1)` and the corresponding element in the second range starting at `first2`. The resulting values are stored in the range starting at `result`. The return value points just beyond the last generated element.
- Example:

```
#include <functional>
#include <vector>
```

```
#include <algorithm>
#include <iostream>
#include <string>
#include <cctype>
#include <iterator>

class Caps
{
    public:
        std::string operator()(std::string const &src)
        {
            std::string tmp = src;

            transform(tmp.begin(), tmp.end(), tmp.begin(), toupper);
            return tmp;
        }
};

using namespace std;

int main()
{
    string words[] = {"alpha", "bravo", "charley"};

    copy(words, transform(words, words + 3, words, Caps()),
                            ostream_iterator<string>(cout, " "));
    cout << endl;

    int         values[] = {1, 2, 3, 4, 5};
    vector<int> squares;

    transform(values, values + 5, values,
                            back_inserter(squares), multiplies<int>());

    copy(squares.begin(), squares.end(),
                            ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    ALPHA BRAVO CHARLEY
    1 4 9 16 25
*/
```

the following differences between the `for_each()` (section 19.1.17) and `transform()` generic algorithms should be noted:

- With `transform()` the *return value* of the function object's `operator()()` member is used; the argument that is passed to the `operator()()` member itself is not changed.

- With `for_each()` the function object's `operator()()` receives a reference to an argument, which itself may be changed by the function object's `operator()()`.

19.1.64 unique()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator unique(ForwardIterator first, ForwardIterator last);`
  - `ForwardIterator unique(ForwardIterator first, ForwardIterator last, BinaryPredicate pred);`
- Description:
  - The first prototype: using `operator==()`, all but the first of consecutively equal elements of the data type to which the iterators point in the range pointed to by `[first, last)` are relocated to the end of the range. The returned forward iterator marks the beginning of the *leftover*. All elements in the range `[first, return-value)` are unique, all elements in the range `[return-value, last)` are equal to elements in the range `[first, return-value)`.
  - The second prototype: all but the first of consecutive elements in the range pointed to by `[first, last)` for which the binary predicate `pred` (expecting two arguments of the data type to which the iterators point) returns `true`, are relocated to the end of the range. The returned forward iterator marks the beginning of the *leftover*. For all pairs of elements in the range `[first, return-value)` `pred` returns `false` (i.e., are *unique*), while `pred` returns `true` for a combination of, as its first operand, an element in the range `[return-value, last)` and, as its second operand, an element in the range `[first, return-value)`.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return !strcasecmp(first.c_str(), second.c_str());
        }
};

using namespace std;

int main()
{
    string words[] = {"alpha", "alpha", "Alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    string *removed = unique(words, words + size);
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    removed = unique(words, words + size, CaseString());
```

```
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    alpha Alpha papa quebec
    Trailing elements are:
    quebec
    alpha papa quebec
    Trailing elements are:
    quebec quebec
*/
```

19.1.65 unique_copy()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result);`
  - `OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator Result, BinaryPredicate pred);`
- Description:
  - The first prototype: the elements in the range `[first, last)` are copied to the resulting container, starting at `result`. Consecutively equal elements (using `operator==()` of the data type to which the iterators point) are copied only once. The returned output iterator points just beyond the last copied element.
  - The second prototype: the elements in the range `[first, last)` are copied to the resulting container, starting at `result`. Consecutive elements in the range pointed to by `[first, last)` for which the binary predicate `pred` returns `true` are copied only once. The returned output iterator points just beyond the last copied element.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <iterator>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return !strcasecmp(first.c_str(), second.c_str());
        }
```

```
};

using namespace std;

int main()
{
    string words[] = {"oscar", "Alpha", "alpha", "alpha",
                                                    "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    vector<string> remaining;

    unique_copy(words, words + size, back_inserter(remaining));

    copy(remaining.begin(), remaining.end(),
            ostream_iterator<string>(cout, " "));
    cout << endl;

    vector<string> remaining2;

    unique_copy(words, words + size,
                           back_inserter(remaining2), CaseString());

    copy(remaining2.begin(), remaining2.end(),
            ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    oscar Alpha alpha papa quebec
    oscar Alpha papa quebec
*/
```

19.1.66 upper_bound()

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, Type const &value);`
  - `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, Type const &value, Compare comp);`
- Description:
  - The first prototype: the sorted elements stored in the iterator range `[first, last)` are searched for the first element that is greater than `value`. The returned iterator marks the first location in the sequence where `value` can be inserted without breaking the sorted order of the elements using `operator<()` of the data type to which the iterators point. If no such element is found, `last` is returned.
  - The second prototype: the elements implied by the iterator range `[first, last)` must have been sorted using the `comp` function or function object. Each element in the range is compared to `value` using the `comp` function. An iterator to the first element for which the

binary predicate `comp`, applied to the elements of the range and `value`, returns `true` is returned. If no such element is found, `last` is returned.

- Example:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int         ia[] = {10, 15, 15, 20, 30};
    size_t    n = sizeof(ia) / sizeof(int);

    cout << "Sequence: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
            *upper_bound(ia, ia + n, 15) << endl;
    cout << "35 can be inserted after " <<
            (upper_bound(ia, ia + n, 35) == ia + n ?
                                "the last element" : "???") << endl;

    sort(ia, ia + n, greater<int>());

    cout << "Sequence: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
            *upper_bound(ia, ia + n, 15, greater<int>()) << endl;
    cout << "35 can be inserted before " <<
            (upper_bound(ia, ia + n, 35, greater<int>()) == ia ?
                                "the first element " : "???") << endl;

    return 0;
}
/*
    Generated output:

    Sequence: 10 15 15 20 30
    15 can be inserted before 20
    35 can be inserted after the last element
    Sequence: 30 20 15 15 10
    15 can be inserted before 10
    35 can be inserted before the first element
*/
```

19.1.67 Heap algorithms

A heap is a kind of binary tree which can be represented by an array. In the standard heap, the key of an element is not smaller than the key of its children. This kind of heap is called a *max heap*. A tree in which numbers are keys could be organized as shown in figure 19.1. Such a tree may also be organized in an array:

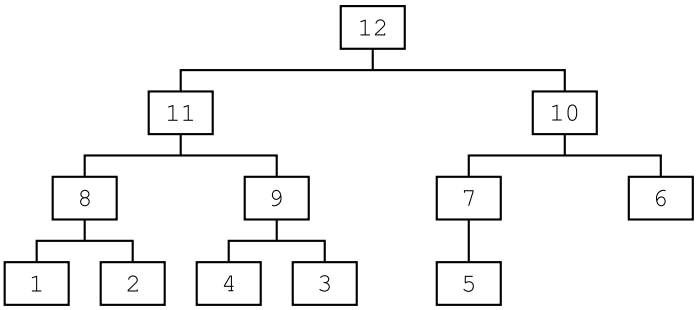

Figure 19.1: A binary tree representation of a heap

```
12, 11, 10, 8, 9, 7, 6, 1, 2, 4, 3, 5
```

In the following description, keep two pointers into this array in mind: a pointer `node` indicates the location of the next node of the tree, a pointer `child` points to the next element which is a child of the `node` pointer. Initially, `node` points to the first element, and `child` points to the second element.

- `*node++ (== 12)`. 12 is the top node. its children are `*child++` (11) and `*child++` (10), both less than 12.
- The next node (`*node++ (== 11)`), in turn, has `*child++` (8) and `*child++` (9) as its children.
- The next node (`*node++ (== 10)`) has `*child++` (7) and `*child++` (6) as its children.
- The next node (`*node++ (== 8)`) has `*child++` (1) and `*child++` (2) as its children.
- Then, node (`*node++ (== 9)`) has children `*child++` (4) and `*child++` (3).
- Finally (as far as children are concerned) (`*node++ (== 7)`) has one child `*child++` (5)

Since `child` now points beyond the array, the remaining nodes have no children. So, nodes 6, 1, 2, 4, 3 and 5 don't have children.

Note that the left and right branches are not ordered: 8 is less than 9, but 7 is larger than 6.

The heap is created by traversing a binary tree level-wise, starting from the top node. The top node is 12, at the zeroth level. At the first level we find 11 and 10. At the second level 6, 7, 8 and 9 are found, etc.

Heaps can be created in containers supporting random access. So, a heap is not, for example, constructed in a list. Heaps can be constructed from an (unsorted) array (using `make_heap()`). The top-element can be pruned from a heap, followed by reordering the heap (using `pop_heap()`), a new element can be added to the heap, followed by reordering the heap (using `push_heap()`), and the elements in a heap can be sorted (using `sort_heap()`, which invalidates the heap, though).

The following subsections show the prototypes of the heap-algorithms, the final subsection provides a small example in which the heap algorithms are used.

19.1.67.1 The 'make_heap()' function

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void make_heap(RandomAccessIterator first, RandomAccessIterator last);`

  - `void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`

- Description:

  - The first prototype: the elements in the range `[first, last)` are reordered to form a max-heap using `operator<()` of the data type to which the iterators point.
  - The second prototype: the elements in the range `[first, last)` are reordered to form a max-heap using the binary comparison function object `comp` to compare elements.

19.1.67.2 The 'pop_heap()' function

- Header file:

```
#include <algorithm>
```

- Function prototypes:

  - `void pop_heap(RandomAccessIterator first, RandomAccessIterator last);`
  - `void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`

- Description:

  - The first prototype: the first element in the range `[first, last)` is moved to `last - 1`. Then, the elements in the range `[first, last - 1)` are reordered to form a max-heap using the `operator<()` of the data type to which the iterators point.
  - The second prototype: the first element in the range `[first, last)` is moved to `last - 1`. Then, the elements in the range `[first, last - 1)` are reordered to form a max-heap using the binary comparison function object `comp` to compare elements.

19.1.67.3 The 'push_heap()' function

- Header file:

```
#include <algorithm>
```

- Function prototypes:

  - `void push_heap(RandomAccessIterator first, RandomAccessIterator last);`
  - `void push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`

- Description:

  - The first prototype: assuming that the range `[first, last - 2)` contains a valid heap, and the element at `last - 1` contains an element to be added to the heap, the elements in the range `[first, last - 1)` are reordered to form a max-heap using the `operator<()` of the data type to which the iterators point.
  - The second prototype: assuming that the range `[first, last - 2)` contains a valid heap, and the element at `last - 1` contains an element to be added to the heap, the elements in the range `[first, last - 1)` are reordered to form a max-heap using the binary comparison function object `comp` to compare elements.

19.1.67.4 The 'sort_heap()' function

- Header file:

```
#include <algorithm>
```

- Function prototypes:
  - `void sort_heap(RandomAccessIterator first, RandomAccessIterator last);`
  - `void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: assuming the elements in the range `[first, last)` form a valid max-heap, the elements in the range `[first, last)` are sorted using `operator<()` of the data type to which the iterators point.
  - The second prototype: assuming the elements in the range `[first, last)` form a valid heap, the elements in the range `[first, last)` are sorted using the binary comparison function object `comp` to compare elements.

19.1.67.5 An example using the heap functions

Here is an example showing the various generic algorithms manipulating heaps:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>

void show(int *ia, char const *header)
{
    std::cout << header << ":\n";
    std::copy(ia, ia + 20, std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
}

using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

    make_heap(ia, ia + 20);
    show(ia, "The values 1-20 in a max-heap");

    pop_heap(ia, ia + 20);
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20);
    show(ia, "Adding 20 (at the end) to the heap again");

    sort_heap(ia, ia + 20);
    show(ia, "Sorting the elements in the heap");


    make_heap(ia, ia + 20, greater<int>());
```

```
    show(ia, "The values 1-20 in a heap, using > (and beyond too)");

    pop_heap(ia, ia + 20, greater<int>());
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20, greater<int>());
    show(ia, "Re-adding the removed element");

    sort_heap(ia, ia + 20, greater<int>());
    show(ia, "Sorting the elements in the heap");

    return 0;
}
/*
    Generated output:

    The values 1-20 in a max-heap:
    20 19 15 18 11 13 14 17 9 10 2 12 6 3 7 16 8 4 1 5
    Removing the first element (now at the end):
    19 18 15 17 11 13 14 16 9 10 2 12 6 3 7 5 8 4 1 20
    Adding 20 (at the end) to the heap again:
    20 19 15 17 18 13 14 16 9 11 2 12 6 3 7 5 8 4 1 10
    Sorting the elements in the heap:
    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
    The values 1-20 in a heap, using > (and beyond too):
    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
    Removing the first element (now at the end):
    2 4 3 8 5 6 7 16 9 10 11 12 13 14 15 20 17 18 19 1
    Re-adding the removed element:
    1 2 3 8 4 6 7 16 9 5 11 12 13 14 15 20 17 18 19 10
    Sorting the elements in the heap:
    20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
*/
```

Chapter 20

Function Templates

C++ supports syntactic constructs allowing programmers to define and use completely general (or abstract) functions or classes, based on generic types and/or (possibly inferred) constant values. In the chapters on abstract containers (chapter 12) and the `STL` (chapter 18) we've already used these constructs, commonly known as the *template mechanism*.

The template mechanism allows us to specify classes and algorithms, fairly independently of the actual types for which the templates will eventually be used. Whenever the template is used, the compiler will generate code, tailored to the particular data type(s) used with the template. This code is generated compile-time from the template's definition. The piece of generated code is called an *instantiation* of the template.

In this chapter the syntactic peculiarities of templates will be covered. The notions of *template type parameter*, *template non-type parameter*, and *function template* will be introduced, and several examples of templates will be offered, both in this chapter and in chapter 23, providing concrete examples of **C++**. Template *classes* are covered in chapter 21.

Templates offered standard by the language already cover containers allowing us to construct both highly complex and standard data structures commonly used in computer science. Furthermore, the `string` (chapter 5) and stream (chapter 6) classes are commonly implemented using templates. So, templates play a central role in present-day **C++**, and should absolutely not be considered an esoteric feature of the language.

Templates should be approached somewhat similarly as generic algorithms: they're a *way of life*; a **C++** software engineer should actively look for opportunities to use them. Initially, templates appear to be rather complex, and you might be tempted to turn your back on them. However, in time their strengths and benefits will be more and more appreciated. Eventually you'll be able to recognize opportunities for using templates. That's the time where your efforts should no longer focus on constructing ordinary functions and classes (i.e., functions or classes that are not templates), but on constructing templates.

This chapter starts by introducing *function templates*. The emphasis is on the required syntax when defining such functions. This chapter lays the foundation upon which the next chapter, introducing class templates and offering several real-life examples, is built.

20.1 Defining function templates

A function template's definition is very similar to the definition of a normal function. A function template has a function head, a function body, a return type, possibly overloaded definitions, etc.. However, different from ordinary functions, function templates always use one or more *formal types*: types for which almost any exising (class or primitive) type could be used. Let's start with a simple example. The following function `add()` expects two arguments, and returns their sum:

```
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
```

Note how closely the above function's definition follows its description: it gets two arguments, and returns its sum. Now consider what would happen if we would have to define this function for, e.g., `int` values. We would have to define:

```
int add(int const &lvalue, int const &rvalue)
{
    return lvalue + rvalue;
}
```

So far, so good. However, were we to add to doubles, we would have to overload this function so that its overloaded version accepts doubles:

```
double add(double const &lvalue, double const &rvalue)
{
    return lvalue + rvalue;
}
```

There is no end to the number of overloaded versions we might be forced to construct: an overloaded version for `std::string`, for `size_t`, for .... In general, we would need an overloaded version for every type supporting `operator+()` and a copy constructor. All these overloaded versions of basically the same function are required because of the strongly typed nature of **C++**. Because of this, a truly generic function cannot be constructed without resorting to the template mechanism.

Fortunately, we've already seen the meat and bones of a template function. Our initial function `add()` actually is an implementation of such a function. However, it isn't a full template definition yet. If we would give the first `add()` function to the compiler, it would produce an error message like:

```
error: 'Type' was not declared in this scope
error: parse error before 'const'
```

And rightly so, as we failed to define `Type`. The error is prevented when we change `add()` into a full template definition. To do this, we look at the function's implementation and decide that `Type` is actually a *formal* typename. Comparing it to the alternate implementations, it will be clear that we could have changed `Type` into `int` to get the first implementation, and into `double` to get the second.

The full template definition allows for this formal character of the `Type` typename. Using the keyword `template`, we prefix one line to our initial definition, obtaining the following function template definition:

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
```

In this definition we distinguish:

- The keyword `template`, starting a template definition or declaration.
- The angle bracket enclosed list following `template`: it is a list, containing one or more comma-separated elements. This angle bracket enclosed list is called the *template parameter list*. When multiple elements are used, it could look like, e.g.,

```
typename Type1, typename Type2
```

- Inside the template parameter list we find the *formal type name* `Type`. It is a formal type name, comparable to a formal parameter name in a function's definition. Up to now we've only encountered formal variable names with functions. The *types* of the parameters were always known by the time the function was defined. Templates escalate the notion of formal names one step further up the ladder, allowing type names to be formalized, rather than just the formal parameter variable names themselves. The fact that `Type` is a formal type name is indicated by the keyword `typename`, prefixed to `Type` in the template parameter list. A formal type name like `Type` is also called a *template type parameter*. Template non-type parameters also exist, and are introduced below.

  Other texts on **C++** sometimes use the keyword `class` where we use `typename`. So, in other texts template definitions might start with a line like:

  ```
  template <class Type>
  ```

  Using `class` instead of `typename` is now, however, considered an anachronism, and is deprecated: a template type parameter is, after all, a type name.

- The function head: it is like a normal function head, albeit that the template's type parameters must be used in its parameter list. When the function is actually called using actual arguments having actual types, these actual types are then used by the compiler to determine which version (overloaded to fit the actual argument types) of the function template must be used. At this point (i.e., where the function is called), the compiler will create the ordinary function, a process called *instantiation*. The function head may also use a formal type to specify its return value. This feature was actually used in the `add()` template's definition.

- The function parameters are specified as `Type const &` parameters. This has the usual meaning: the parameters are references to `Type` objects or values that will not be modified by the function.

- The function body: it is like a normal function body. In the body the formal type names may be used to define or declare variables, which may then be used as any other local variable. Even so, there are some restrictions. Looking at `add()`'s body, it is clear that `operator+()` is used, as well as a copy constructor, as the function returns a value. This allows us to formulate the following restrictions for the formal type `Type`:

  - `Type` should support `operator+()`
  - `Type` should support a copy constructor

  Consequently, while `Type` could be a `std::string`, it could never be an `ostream`, as neither `operator+()` nor the copy constructor are available for streams.

Normal scope rules and identifier visibility rules apply to template definitions. Formal typenames overrule, within the template definition's scope, any identifiers having identical names having wider scopes.

Look again at the function's parameters, as defined in its parameter list. By specifying `Type const &` rather than `Type` superfluous copying is prevented, at the same time allowing values of primitive types to be passed as arguments to the function. So, when `add(3, 4)` is called, `int(4)` will be assigned to `Type const &rvalue`. In general, function parameters should be defined as `Type const &` to prevent unnecessary copying. The compiler is smart enough to handle 'references to references' in this case, which is something the language normally does not support. For example, consider the following `main()` function (here and in the following simple examples it is assumed that the template and the required headers and namespace declarations have been provided):

```
int main()
{
    size_t const &uc = size_t(4);
    cout << add(uc, uc) << endl;
}
```

Here uc is a reference to a constant size_t. It is passed as argument to add(), thereby initializing lvalue and rvalue as Type const & to size_t const & values, with the compiler interpreting Type as size_t. Alternatively, the parameters might have been specified using Type &, rather than Type const &. The disadvantage of this (non-const) specification being that temporary values cannot be passed to the function anymore. The following will fail to compile:

```
int main()
{
    cout << add(string("a"), string("b")) << endl;
}
```

Here, a string const & cannot be used to initialize a string &. On the other hand, the following *will* compile, with the compiler deciding that Type should be considered a string const:

```
int main()
{
    string const &s = string("a");
    cout << add(s, s) << endl;
}
```

What can we deduce from these examples?

- In general, function parameters should be specified as Type const & parameters to prevent unnecessary copying.
- The template mechanism is fairly flexible, in that it will interpret formal types as plain types, const types, pointer types, etc., depending on the actually provided types. The rule of thumb is that the formal type is used as a generic mask for the actual type, with the formal type name covering whatever part of the actual type must be covered. Some examples, assuming the parameter is defined as Type const &:

argument type	**Type ==**
size_t const	size_t
size_t	size_t
size_t *	size_t *
size_t const *	size_t const *

As a second example of a function template, consider the following function definition:

```
template <typename Type, size_t Size>
Type sum(Type const (&array)[Size])
{
    Type t = Type();

    for (size_t idx = 0; idx < Size; idx++)
        t += array[idx];

    return t;
}
```

This template definition introduces the following new concepts and features:

- Its template parameter list has two elements. Its first element is a well-known template type parameter, but its second element has a very specific type: a size_t. Template parameters of specific (i.e., non-formal) types used in template parameter lists are called *template non-type parameters*. A template *non-type parameter* represents a constant expression, which must be known by the time the template is instantiated, and which is specified in terms of existing types, such as a size_t.

- Looking at the function's head, we see one parameter:

  ```
  Type const (&array)[Size]
  ```

  This parameter defines `array` as a reference parameter to an array having `Size` elements of type `Type`, that may not be modified.

- In the parameter definition, both `Type` and `Size` are used. `Type` is of course the template's type parameter `Type`, but `Size` is also a template parameter. It is a `size_t`, whose value must be inferable by the compiler when it compiles an actual call of the `sum()` function template. Consequently, `Size` must be a `const` value. Such a constant expression is called a *template non-type parameter*, and it is named in the template's parameter list.

- When the function template is called, the compiler must be able to infer not only `Type`'s concrete value, but also `Size`'s value. Since the function `sum()` only has one parameter, the compiler is only able to infer `Size`'s value from the function's actual argument. It can do so if the provided argument is an array (of known and fixed size), rather than a pointer to `Type` elements. So, in the following `main()` function the first statement will compile correctly, whereas the second statement won't:

  ```
  int main()
  {
      int values[5];
      int *ip = values;

      cout << sum(values) << endl;    // compiles OK
      cout << sum(ip) << endl;        // won't compile
  }
  ```

- Inside the function, the statement `Type t = Type()` is used to initialize `t` to a default value. Note here that no fixed value (like 0) is used. Any type's default value may be obtained using its default constructor, rather than using a fixed numeric value. Of course, not every class accepts a numeric value as an argument to one of its constructors. But all types, even the primitive types, support default constructors (actually, some classes do not implement a default constructor, or make it inaccessible; but most do). The default constructor of primitive types will initialize their variables to 0 (or `false`). Furthermore, the statement `Type t = Type()` is a true initialization: `t` is initialized by `Type`'s default constructor, rather than using `Type`'s copy constructor to assign `Type()`'s copy to `t`.

  It's interesting to note here (although unrelated to the current topic) that the syntactic construction `Type t(Type())` *cannot* be used, even though it also looks like a proper initialization. Usually an initializing argument can be provided to an object's definition, like `string s("hello")`. Why, then is `Type t = Type()` accepted, whereas `Type t(Type())` isn't? When `Type t(Type())` is used, it won't even be clear at the site of the definition that it's not a `Type` object's default initialization. Instead, the compiler will only start generating error messages once `t` is used. This is caused by the fact that in **C++** (and in **C** alike) the compiler will try to see a function or function pointer whenever possible: the *function prevalence rule*. According to this rule `Type()` will be interpreted as a *pointer to a function* expecting no arguments and returning a `Type`, unless the compiler clearly is unable to do so. In the initialization `Type t = Type()` it can't see a pointer to a function, as a `Type` object cannot be given the value of a function pointer (remember: `Type()` is interpreted as `Type (*)()` whenever possible). But in `Type t(Type())` it can use the pointer interpretation: `t` is now *declared* as a *function* expecting a pointer to a function returning a `Type`, with `t` itself also returning a `Type`. E.g., `t` could have been defined as:

  ```
  Type t(Type (*tp)())
  {
      return (*tp)();
  }
  ```

- Comparable to the first function template, `sum()` also assumes the existence of certain public members in `Type`'s class. This time `operator+=()` and `Type`'s copy constructor.

Like class definitions, template definitions should not contain `using` directives or declarations: the template might be used in a situation where such a directive overrides the programmer's intentions: ambiguities or other conflicts may result from the template's author and the programmer using different `using` directives (E.g, a `cout` variable defined in the `std` namespace and in the programmer's own namespace). Instead, within template definitions only fully qualified names, including all required namespace specifications should be used.

20.1.1 Alternate function template syntax (C++0x)

Traditional **C++** requires function templates to specify their return type or to specify the return type as a template type parameter. Consider the following function:

```
int add(int lhs, int rhs)
{
    return lhs + rhs;
}
```

The above function may be converted to a (more generic) template function:

```
template <typename Lhs, typename Rhs>
Lhs add(Lhs lhs, Rhs rhs)
{
    return lhs + rhs;
}
```

Unfortunately, when the template function is called as

```
add(3, 3.4)
```

the intended return type is probably a `double` rather than an `int`. This can be solved by adding an additional template type parameter specifying the return type, but it requires the specification of that type:

```
add<double>(3, 3.4);
```

This problem can't be solved using `decltype` (cf. section 3.3.5) as `lhs` and `rhs` aren't known to the compiler by the time `decltype` is used in the following attempt to get rid of the additional type parameter:

```
template <typename Lhs, typename Rhs>
decltype(lhs + rhs) add(Lhs lhs, Rhs rhs)
{
    return lhs + rhs;
}
```

The `decltype`-based definition of a function's return type may become fairly complex. To reduce the complexities the C++0x standard provides the *late-specified return type* syntax that *does* allow the use of `decltype` to define a function's return type (it is primarily used with template functions but it may also be used for non-template functions):

```
template <typename Lhs, typename Rhs>
auto add(Lhs lhs, Rhs rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}
```

When this function is used in a statement like `cout « add(3, 3.4)` the resulting value will be 6.4, which is most likely the intended result, rather than 6. As an example how a late-specified return type might reduce the complexity of a function's return type definition consider the following:

```
template <class T, class U>
decltype((*(T*)0)+(*(U*)0)) add(T t, U u);
```

Using a late-specified return type we get the equivalent:

```
template <class T, class U>
auto add(T t, U u) -> decltype(t+u);
```

which is arguably easier to read.

The expression specified with `decltype` does not necessarily use the parameters `lhs` and `rhs`. In the following function definition `lhs.length()` is used instead of `lhs` itself:

```
template <typename Class, typename Rhs>
auto  add(Class lhs, Rhs rhs) -> decltype(lhs.length() + rhs)
{
    return lhs.length() + rhs;
}
```

Any variable visible at the time `decltype` is compiled can be used in the `decltype` expression. However, it is currently not possible to handle member selection through pointers to members. The following code aims at specifying the address of a member function as `add`'s first argument and then use its return value type to determine the template function's return type:

```
std::string global;

template <typename MEMBER, typename RHS>
auto  add(MEMBER mem, RHS rhs) -> decltype((global.*mem)() + rhs)
{
    return (global.*mem)() + rhs;
}
```

Although the above function compiles fine, it cannot currently be used as it results in a compiler error message like *unimplemented: mangling dotstar_expr* (generated for a statement like `cout « add(&string::length, 3.4)`).

20.2 Reference Wrappers (C++0x)

Situations exist where the compiler is unable to infer that a reference rather than a value is passed to a template function. In the following example the template function `outer` receives `int x` as its argument and the compiler will dutifully infer that `Type` is `int`:

Compilation will of course fail and the compiler nicely reports the inferred type, e.g.:

```
In function 'void outer(Type) [with Type = int]': ...
```

Unfortunately the same happens too when in the next example `call` is used. The function `call` is a template expecting a function that takes an argument which is then itself modified and a value to pass on to that function. Such a function is, e.g., `sqrtArg` expecting a reference to a `double`, which is modified by calling `std::sqrt`.

Assuming `double value = 3` then `call(sqrtArg, value)` will not modify `value` as the compiler infers `Arg` to be `double` and hence passes `value` by value.

To have `value` itself changed the compiler must be informed that `value` must be passed by reference. Note that it might not be acceptable to define `call`'s template argument as `Arg &` as *not* changing the actual argument might be appropriate in some situations.

The C++0x standard offers the `ref(arg)` and `cref(arg)` *reference wrappers* that will take an argument and return it as a reference-typed argument. To actually change `value` it can be passed to `call` using `ref(value)` as shown in the following `main` function:

To use a reference wrapper the header file `functional` must be included.

20.3 Argument deduction

In this section we'll concentrate on the process by which the compiler deduces the actual types of the template type parameters when a template function is called, a process called *template parameter deduction*. As we've already seen, the compiler is able to substitute a wide range of actual types for a single formal template type parameter. Even so, not every thinkable conversion is possible. In particular when a function has multiple parameters of the same template type parameter, the compiler is very restrictive in what argument types it will actually accept.

When the compiler deduces the actual types for template type parameters, it will only consider the types of the arguments. Neither local variables nor the function's return value is considered in this process. This is understandable: when a function is called, the compiler will only see the function template's arguments with certainty. At the point of the call it will definitely not see the types of the function's local variables, and the function's return value might not actually be used, or may be assigned to a variable of a subrange (or super-range) type of a deduced template type parameter. So, in the following example, the compiler won't ever be able to call `fun()`, as it has no way to deduce the actual type for the `Type` template type parameter.

```
template <typename Type>
Type fun()              // can never be called as 'fun()'
{
    return Type();
}
```

Although the compiler won't be able to handle a call to '`fun()`', it *is* possible to call `fun()` using an explicit type specification. E.g., `fun<int>()` will call `fun()`, instantiated for `int`. This, of course is *not* the same as *compiler* argument deduction.

In general, when a function has multiple parameters of identical template type parameters, the actual types must be exactly the same. So, whereas

```
void binarg(double x, double y);
```

may be called using an `int` and a `double`, with the `int` argument implicitly being converted to a `double`, the corresponding function template cannot be called using an `int` and `double` argument: the compiler won't itself promote `int` to `double` and to decide next that `Type` should be `double`:

```
template <typename Type>
void binarg(Type const &p1, Type const &p2)
{}

int main()
{
    binarg(4, 4.5); // ?? won't compile: different actual types
}
```

What, then, are the transformations the compiler will apply when deducing the actual types of template type parameters? It will perform only three types of parameter type transformations (and a fourth one to function parameters of any fixed type (i.e., of a function non-template parameter type)). If it cannot deduce the actual types using these transformations, the template function will not be considered. These transformations are:

- *lvalue transformations*, creating an *rvalue* from an *lvalue*;
- *qualification transformations*, inserting a `const` modifier to a non-constant argument type;
- *transformation to a base class instantiated from a class template*, using a template base class when an argument of a template derived class type was provided in the call.
- Standard transformations for template non-type function parameters. This isn't a template parameter type transformation, but it refers to any remaining template non-type parameter of function templates. For these function parameters the compiler will perform any standard conversion it has available (e.g., `int` to `size_t`, `int` to `double`, etc.).

The first three types of transformations will now be discussed and illustrated.

20.3.1 Lvalue transformations

There are three types of *lvalue transformations*:

- **lvalue-to-rvalue transformations.**

  An lvalue-to-rvalue transformation is applied when an `rvalue` is required, and an `lvalue` is provided. This happens when a variable is used as argument to a function specifying a *value parameter*. For example,

```
template<typename Type>
Type negate(Type value)
{
    return -value;
}
int main()
{
    int x = 5;
    x = negate(x);  // lvalue (x) to rvalue (copies x)
}
```

- **array-to-pointer transformations.**

  An array-to-pointer transformation is applied when the name of an array is assigned to a pointer variable. This is frequently seen with functions defining pointer parameters. When calling such functions, arrays are often specified as their arguments. The array's address is then assigned to the pointer-parameter, and its type is used to deduce the corresponding template parameter's type. For example:

```
template<typename Type>
Type sum(Type *tp, size_t n)
{
    return accumulate(tp, tp + n, Type());
}
int main()
{
    int x[10];
    sum(x, 10);
}
```

In this example, the location of the array `x` is passed to `sum()`, expecting a pointer to some type. Using the array-to-pointer transformation, `x`'s address is considered a pointer value which is assigned to `tp`, deducing that `Type` is `int` in the process.

- **function-to-pointer transformations.**

  This transformation is most often seen with function templates defining a parameter which is a pointer to a function. When calling such a function the name of a function may be specified as its argument. The address of the function is then assigned to the pointer-parameter, deducing the template type parameter in the process. This is called a function-to-pointer transformation. For example:

```
#include <cmath>

template<typename Type>
void call(Type (*fp)(Type), Type const &value)
{
    (*fp)(value);
}
int main()
{
    call(&sqrt, 2.0);
}
```

  In this example, the address of the `sqrt()` function is passed to `call()`, expecting a pointer to a function returning a `Type` and expecting a `Type` for its argument. Using the function-to-pointer transformation, `sqrt`'s address is considered a pointer value which is assigned to `fp`, deducing that `Type` is `double` in the process. Note that the argument `2.0` could not have been specified as `2`, as there is no `int sqrt(int)` prototype. Also note that the function's first parameter specifies `Type (*fp)(Type)`, rather than `Type (*fp)(Type const &)` as might have been expected from our previous discussion about how to specify the types of function template's parameters, preferring references over values. However, `fp`'s argument `Type` is not a function template parameter, but a parameter of the function `fp` points to. Since `sqrt()` has prototype `double sqrt(double)`, rather than `double sqrt(double const &)`, `call()`'s parameter `fp` *must* be specified as `Type (*fp)(Type)`. It's that strict.

20.3.2 Qualification transformations

A *qualification transformation* adds `const` or `volatile` qualifications to *pointers*. This transformation is applied when the function template's parameter is explicitly defined using a `const` (or `volatile`) modifier, and the function's argument isn't a `const` or `volatile` entity. In that case, the transformation adds `const` or `volatile`, and subsequently deduces the template's type parameter. For example:

```
template<typename Type>
Type negate(Type const &value)
{
    return -value;
}
int main()
{
    int x = 5;
    x = negate(x);
}
```

Here we see the function template's `Type const &value` parameter: a reference to a `const Type`. However, the argument isn't a `const int`, but an `int` that can be modified. Applying a qualification transformation, the compiler adds `const` to `x`'s type, and so it matches `int const x` with `Type const &value`, deducing that `Type` must be `int`.

20.3.3 Transformation to a base class

Although the *construction* of class templates is the topic of chapter 21, class templates have already extensively been *used* earlier. For example, abstract containers (covered in chapter 12) are actually defined as class templates. Class templates can, like ordinary classes, participate in the construction of class hierarchies. In section 21.10 it is shown how a class template can be derived from another class template.

As class template derivation remains to be covered, the following discussion is necessarily somewhat abstract. Optionally, the reader may of course skip briefly to section 21.10, and return back to this section thereafter.

In this section it should be assumed, for the sake of argument, that a class template `Vector` has somehow been derived from a `std::vector`. Furthermore, assume that the following function template has been constructed to sort a vector using some function object `obj`:

```
template <typename Type, typename Object>
void sortVector(std::vector<Type> vect, Object const &obj)
{
    sort(vect.begin(), vect.end(), obj);
}
```

To sort `std::vector<string>` objects case-insensitively, the `class Caseless` could be constructed as follows:

```
class CaseLess
{
    public:
        bool operator()(std::string const &before,
                        std::string const &after) const
        {
            return strcasecmp(before.c_str(), after.c_str()) < 0;
        }
};
```

Now various vectors may be sorted using `sortVector()`:

```
int main()
{
    std::vector<string> vs;
    std::vector<int> vi;

    sortVector(vs, CaseLess());
    sortVector(vi, less<int>());
}
```

Applying the transformation *transformation to a base class instantiated from a class template*, the function template `sortVectors()` may now also be used to sort `Vector` objects. For example:

```
int main()
{
    Vector<string> vs;      // 'Vector' instead of 'std::vector'
    Vector<int> vi;

    sortVector(vs, CaseLess());
    sortVector(vi, less<int>());
}
```

In this example, `Vectors` were passed as argument to `sortVector()`. Applying the transformation to a base class instantiated from a class template, the compiler will consider `Vector` to be a `std::vector`, and is thus able to deduce the template's type parameter. A `std::string` for the `Vector vs`, an `int` for `Vector vi`.

Please note that the purpose of the various template parameter type deduction transformations is *not* to match function arguments to function parameters, but rather, having matched arguments to parameters, to determine the *actual types* of the various template type parameters.

20.3.4 The template parameter deduction algorithm

The compiler uses the following algorithm to deduce the actual types of its template type parameters:

- In turn, the function template's parameters are identified using the arguments of the called function.
- For each template parameter used in the function template's parameter list, the template type parameter is matched with the corresponding argument's type (e.g., `Type` is `int` if the argument is `int x`, and the function's parameter is `Type &value`).
- While matching the argument types to the template type parameters, the three allowed transformations (see section 20.3) for template type parameters are applied where necessary.
- If identical template type parameters are used with multiple function parameters, the deduced template types must be exactly the same. So, the next function template cannot be called with an `int` and a `double` argument:

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
```

  When calling this function template, two identical types must be used (albeit that the three standard transformations are of course allowed). If the template deduction mechanism does not come up with identical actual types for identical template types, then the function template will not be instantiated.

20.4 Declaring function templates

Up to now, we've only defined function templates. There are various consequences of including function template definitions in multiple source files, none of them serious, but worth knowing.

- Like class interfaces, template definitions are usually included in header files. Every time a header file containing a template definition is read by the compiler, the compiler must process the full definition, even though it might not actually need the template. This will relatively slow-down the compilation. For example, compiling a template header file like `algorithm` on my old laptop takes about four times the amount of time it takes to compile a plain header file like `cmath`. The header file `iostream` is even harder to process, requiring almost 15 times the amount of time it takes to process `cmath`. Clearly, processing templates is serious business for the compiler. On the other hand: don't overweigh this drawback: compilers are getting better and better in their template processing capacity and computers keep on getting faster and faster. What was a nuisance a few years ago is hardly noticeable today.
- Every time a function template is instantiated, its code appears in the resulting object module. However, if multiple instantiations of a template using the same actual types for its template parameter exist in multiple object files, then the linker will weed out superfluous instantiations.

In the final program only one instantiation for a particular set of actual template type parameters will be used (see also section 20.5 for an illustration). Therefore, the linker will have an additional task to perform (*viz.* weeding out multiple instantiations), which will slow down the linking process.

- Sometimes the definitions themselves are not required, but only references or pointers to the templates are required. Requiring the compiler to process the full template definitions in those cases will unnecessarily slow down the compilation process.
- In the context of *template meta programming* (see chapter 22) it is sometimes not even required to provide a template implementation. Instead, only *specializations* (cf. section 20.8) are created, all of which are then based on the mere *declaration*.

So, depending on the context, template definitions may not be required. Usually template definitions do exist, but when appropriate the software enginieer may opt to *declare* a template, rather than to include its definition time and again in various source files.

When templates are declared, the compiler will not have to process the template's definitions again and again; and no instantiations will be created on the basis of template declarations alone. Any actually required instantiation must then be available elsewhere (of course, this holds true for declarations in general). Unlike the situation we encounter with ordinary functions, which are usually stored in libraries, it is currently not possible to store templates in libraries (although the compiler may construct *precompiled header files*). Consequently, using template declarations puts a burden on the shoulders of the software engineer, who has to make sure that the required instantiations exist. Below a simple way to accomplish that is introduced.

A function template declaration is simply created: the function's body is replaced by a semicolon. Note that this is exactly identical to the way ordinary function declarations are constructed. So, the previously defined function template `add()` can simply be declared as

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue);
```

Actually, we've already encountered template declarations. The header file `iosfwd` may be included in sources not requiring instantiations of elements from the class `ios` and its derived classes. For example, in order to compile the *declaration*

```
std::string getCsvline(std::istream &in, char const *delim);
```

it is not necessary to include the `string` and `istream` header files. Rather, a single

```
#include <iosfwd>
```

is sufficient, requiring about one-ninth the amount of time it takes to compile the declaration when `string` and `istream` are included.

20.4.1 Instantiation declarations

So, if declaring function templates speeds up the compilation and the linking phases of a program, how can we make sure that the required instantiations of the function templates will be available when the program is eventually linked together?

For this a variant of a declaration is available, a so-called *explicit instantiation declaration*. An explicit instantiation declaration contains the following elements:

- It starts with the keyword `template`, omitting the template parameter list.

- Next the function's return type and name are specified.
- The function name is followed by a *type specification list*, a list of types between angle brackets, each type specifying the actual type of the corresponding template type parameter in the template's parameter list.
- Finally the function's parameter list is specified, terminated by a semicolon.

Although this is a declaration, it is actually understood by the compiler as a request to instantiate that particular variant of the function.

Using explicit instantiation declarations all instantiations of template functions required by a program can be collected in one file. This file, which should be a normal *source* file, should include the template definition header file, and should next specify the required instantiation declarations. Since it's a source file, it will not be included by other sources. So namespace `using` directives and declarations may safely be used once the required headers have been included. Here is an example showing the required instantiations for our earlier `add()` template, instantiated for `double`, `int`, and `std::string` types:

```
#include "add.h"
#include <string>
using namespace std;

template int add<int>(int const &lvalue, int const &rvalue);
template double add<double>(double const &lvalue, double const &rvalue);
template string add<string>(string const &lvalue, string const &rvalue);
```

If we're sloppy and forget to mention an instantiation required by our program, then the repair can easily be made: just add the missing instantiation declaration to the above list. After recompiling the file and relinking the program we're done.

20.5 Instantiating function templates

A template is not instantiated when its definition is read by the compiler. A template is merely a *recipe* telling the compiler how to create particular code once it's time to do so. It's very much like a recipe in a cooking book: you reading a cake's recipe doesn't mean you have actually cooked that cake by the time you've read the recipe.

So, when is a function template actually instantiated? There are two situations in which the compiler will decide to instantiate templates:

- They are instantiated when they're actually used (e.g., the function `add()` is called with a pair of `size_t` values);
- When addresses of function templates are taken they are instantiated. For example:

```
#include "add.h"

char (*addptr)(char const &, char const &) = add;
```

The location of statements causing the compiler to instantiate a template is called the template's *point of instantiation*. The point of instantiation has serious implications for the function template's code. These implications are discussed in section 20.11.

The compiler is not always able to deduce the template's type parameters unambiguously. In that case the compiler reports an ambiguity which must be solved by the software engineer. Consider the following code:

```
#include <iostream>
```

```
#include "add.h"

size_t fun(int (*f)(int *p, size_t n));
double fun(double (*f)(double *p, size_t n));

int main()
{
    std::cout << fun(add) << std::endl;
}
```

When this little program is compiled, the compiler reports an ambiguity it cannot resolve. It has two candidate functions, as for each overloaded version of `fun()` a proper instantiation of `add()` can be constructed:

```
error: call of overloaded 'fun(<unknown type>)' is ambiguous
note: candidates are: int fun(size_t (*)(int*, size_t))
note:                 double fun(double (*)(double*, size_t))
```

Situations like these should of course be avoided. Function templates can only be instantiated if there's no ambiguity. Ambiguities arise when multiple functions emerge from the compiler's function selection mechanism (see section 20.10). It is up to us to resolve these ambiguities. Ambiguities like the above can be resolved using a blunt `static_cast` (as we select among alternatives, all of them possible and available):

```
#include <iostream>
#include "add.h"

int fun(int (*f)(int const &lvalue, int const &rvalue));
double fun(double (*f)(double const &lvalue, double const &rvalue));

int main()
{
    std::cout << fun(
                    static_cast<int (*)(int const &, int const &)>(add)
                ) << std::endl;
    return 0;
}
```

But if possible, type casts should be avoided. How to avoid casts in situations like these is explained in the next section (20.6).

As mentioned in section 20.4, the linker will remove identical instantiations of a template from the final program, leaving only one instantiation for each unique set of actual template type parameters. Let's have a look at an example showing this behavior of the linker. To illustrate the linker's behavior, we will do as follows:

- First we construct several source files:
  - `source1.cc` defines a function `fun()`, instantiating `add()` for `int`-type arguments, including `add()`'s template definition. It displays `add()`'s address. Here is `source1.cc`:

```
union PointerUnion
{
    int (*fp)(int const &, int const &);
    void *vp;
};

#include <iostream>
#include "add.h"
```

```
#include "pointerunion.h"

void fun()
{
    PointerUnion pu = { add };

    std::cout << pu.vp << std::endl;
}
```

- `source2.cc` defines the same function, but only declares the proper `add()` template using a template declaration (*not* an instantiation declaration). Here is `source2.cc`:

```
#include <iostream>
#include "pointerunion.h"

template<typename Type>
Type add(Type const &, Type const &);

void fun()
{
    PointerUnion pu = { add };

    std::cout << pu.vp << std::endl;
}
```

- `main.cc` again includes `add()`'s template definition, declares the function `fun()` and defines `main()`, defining `add()` for `int`-type arguments as well and displaying `add()`'s function address. It also calls the function `fun()`. Here is `main.cc`:

```
#include <iostream>
#include "add.h"
#include "pointerunion.h"

void fun();

int main()
{
    PointerUnion pu = { add };

    fun();
    std::cout << pu.vp << std::endl;
}
```

- All sources are compiled to object modules. Note the different sizes of `source1.o` (2104 bytes using `g++` version 4.1.2. All sizes reported here may differ somewhat for different compilers and/or run-time libraries) and `source2.o` (1928 bytes). Since `source1.o` contains the instantiation of `add()`, it is somewhat larger than `source2.o`, containing only the template's declaration. Now we're ready to start our little experiment.

- Linking `main.o` and `source1.o`, we obviously link together two object modules, each containing its own instantiation of the same template function. The resulting program produces the following output:

```
0x80486d8
0x80486d8
```

  Furthermore, the size of the resulting program is 8701 bytes.

- Linking `main.o` and `source2.o`, we now link together an object module containing the instantiation of the `add()` template, and another object module containing the mere declaration of the same template function. So, the resulting program cannot but contain a single instantiation of the required function template. This program has exactly the same size, and produces exactly the same output as the first program.

So, from our little experiment we can conclude that the linker will indeed remove identical template instantiations from a final program, and that using mere template declarations will not result in template instantiations.

20.6 Using explicit template types

In the previous section (section 20.5) we've seen that the compiler may encounter ambiguities when attempting to instantiate a template. We've seen an example in which overloaded versions of a function `fun()` existed, expecting different types of arguments, both of which could have been provided by an instantiation of a function template. The intuitive way to solve such an ambiguity is to use a `static_cast`, but as noted: if possible, casts should be avoided.

When function templates are involved, such a `static_cast` may indeed neatly be avoided using *explicit template type arguments*. When explicit template type arguments are used the compiler is explicitly informed about the actual template type parameters it should use when instantiating a template. Here, the function's name is followed by an *actual template parameter type list* which may again be followed by the function's argument list, if required. The actual types mentioned in the actual template parameter list are used by the compiler to 'deduce' the actual types of the corresponding template types of the function's template parameter type list. Here is the same example as given in the previous section, now using explicit template type arguments:

```
#include <iostream>
#include "add.h"

int fun(int (*f)(int const &lvalue, int const &rvalue));
double fun(double (*f)(double const &lvalue, double const &rvalue));

int main()
{
    std::cout << fun(add<int>) << std::endl;
    return 0;
}
```

Explicit template argument types can be used in situations where the compiler has no way to detect which types should actually be used. E.g., in section 20.3 the function template `Type fun()` was defined. To instantiate this function for the `double` type, we can use `fun<double>()`.

20.7 Overloading function templates

Let's once again look at our `add()` template. That template was designed to return the sum of two entities. If we would want to compute the sum of three entities, we could write:

```
int main()
{
    add(2, add(3, 4));
}
```

This is a perfectly acceptable solution for the occasional situation. However, if we would have to add three entities regularly, an *overloaded* version of the `add()` function, expecting three arguments, might be a useful thing to have. There's a simple solution to this problem: function templates may be overloaded.

To define an overloaded version, merely put multiple definitions of the template in its definition header file. So, with the `add()` function this would boil down to, e.g.:

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}

template <typename Type>
Type add(Type const &lvalue, Type const &mvalue, Type const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

The overloaded function does not have to be defined in terms of simple values. Like all overloaded functions, just a unique set of function parameters is enough to define an overloaded version. For example, here's an overloaded version that can be used to compute the sum of the elements of a vector:

```
template <typename Type>
Type add(std::vector<Type> const &vect)
{
    return accumulate(vect.begin(), vect.end(), Type());
}
```

Overloading templates does not have to restrict itself to the function's parameter list. The template's type parameter list itself may also be overloaded. The last definition of the `add()` template allows us to specify a `std::vector` as its first argument, but no `deque` or `map`. Overloaded versions for those types of containers could of course be constructed, but where's the end to that? Instead, let's look for common characteristics of these containers, and if found, define an overloaded function template on these common characteristics. One common characteristic of the mentioned containers is that they all support `begin()` and `end()` members, returning iterators. Using this, we could define a template type parameter representing containers that must support these members. But mentioning a plain 'container type' doesn't tell us for what data type it has been instantiated. So we need a second template type parameter representing the container's data type, thus overloading the template's type parameter list. Here is the resulting overloaded version of the `add()` template:

```
template <typename Container, typename Type>
Type add(Container const &cont, Type const &init)
{
    return std::accumulate(cont.begin(), cont.end(), init);
}
```

One may wonder whether the `init` parameter could not be left out of the parameter list as `init` will often have a default initialization value. The answer is a somewhat complex 'yes', It *is* possible to define the `add()` function as follows:

```
template <typename Type, typename Container>
Type add(Container const &cont)
{
    Type init = Type();
    return std::accumulate(cont.begin(), cont.end(), init);
}
```

But note that the template's type parameters were reordered, which is necessary because the compiler won't be able to determine `Type` in a call like:

```
int x = add(vectorOfInts);
```

However, it is also possible to use a third kind of template parameter, a *template template parameter*, which *does* allow the compiler to determine `Type` directly from the actual container argument used when calling `add()`. Template template parameters are discussed in section 22.3.

With all these overloaded versions in place, we may now start the compiler to compile the following function:

```
using namespace std;

int main()
{
    vector<int> v;

    add(3, 4);          // 1 (see text)
    add(v);             // 2
    add(v, 0);          // 3
}
```

- With the first statement, the compiler recognizes two identical types, both `int`. It will therefore instantiate `add<int>()`, our very first definition of the `add()` template.
- With statement two, a single argument is used. Consequently, the compiler will look for an overloaded version of `add()` requiring but one argument. It finds the version expecting a `std::vector`, deducing that the template's type parameter must be `int`. It instantiates

  ```
  add<int>(std::vector<int> const &)
  ```

- With statement three, the compiler again encounters an argument list having two arguments. However, the types of the arguments are different, so it cannot use the `add()` template's first definition. But it *can* use the last definition, expecting entities having different types. As a `std::vector` supports `begin()` and `end()`, the compiler is now able to instantiate the function template

  ```
  add<std::vector<int>, int>(std::vector<int> const &, int const &)
  ```

Having defined `add()` using two different template type parameters, and a function template having a parameter list containing two parameters of these types, we've exhausted the possibilities to define an `add()` function template having a function parameter list showing two different types. Even though the parameter types are different, we're still able to define a function template `add()` as a function template merely returning the sum of two differently typed entities:

```
template <typename T1, typename T2>
T1 add(T1 const &lvalue, T2 const &rvalue)
{
    return lvalue + rvalue;
}
```

However, now we won't be able to instantiate `add()` using two differently typed arguments anymore: the compiler won't be able resolve the ambiguity. It cannot choose which of the two overloaded versions defining two differently typed function parameters to use:

```
int main()
{
    add(3, 4.5);
}
/*
    Compiler reports:
```

```
        error: call of overloaded ‘add(int, double)’ is ambiguous
        error: candidates are: Type add(const Container&, const Type&)
                                    [with Container = int, Type = double]
        error:                 T1 add(const T1&, const T2&)
                                    [with T1 = int, T2 = double]
    */
```

Consider once again the overloaded function accepting three arguments:

```
template <typename Type>
Type add(Type const &lvalue, Type const &mvalue, Type const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

It may be considered as a disadvantage that only equally typed arguments are accepted by this function: e.g., three `int`s, three `double`s or three `string`s. To remedy this, we define yet another overloaded version of the function, this time accepting arguments of any type. Of course, when calling this function we must make sure that `operator+()` is defined between them, but apart from that there appears to be no problem. Here is the overloaded version accepting arguments of any type:

```
template <typename Type1, typename Type2, typename Type3>
Type1 add(Type1 const &lvalue, Type2 const &mvalue, Type3 const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

Now that we've defined these two overloaded versions, let's call `add()` as follows:

```
    add(1, 2, 3);
```

In this case, one might expect the compiler to report an ambiguity. After all, the compiler might select the former function, deducing that `Type == int`, but it might also select the latter function, deducing that `Type1 == int, Type2 == int` and `Type3 == int`. However, the compiler reports no ambiguity. The reason for this is the following: if an overloaded template function is defined using *more specialized* template type parameters (e.g., all equal types) than another (overloaded) function, for which more general template type parameters (e.g., all different) have been used, then the compiler will select the more specialized function over the more general function wherever possible.

As a rule of thumb: when overloaded versions of a function template are defined, each overloaded version must use a unique combination of template type parameters to avoid ambiguities when the templates are instantiated. Note that the *ordering* of template type parameters in the function's parameter list is not important. When trying to instantiate the following `binarg()` template, an ambiguity will occur:

```
template <typename T1, typename T2>
void binarg(T1 const &first, T2 const &second)
{}
// and:
template <typename T1, typename T2>
void binarg(T2 const &first, T1 const &second)  // exchange T1 and T2
{}
```

The ambiguity should come as no surprise. After all, template type parameters are just formal names. Their names (`T1`, `T2` or `Whatever`) have no concrete meanings whatsoever.

Finally, overloaded functions may be declared, either using plain declarations or instantiation declarations, and explicit template parameter types may also be used. For example:

- Declaring a function template `add()` accepting containers of a certain type:

```
template <typename Container, typename Type>
Type add(Container const &container, Type const &init);
```

- The same function, but now using an instantiation declaration (note that this requires that the compiler has already seen the template's definition):

```
template int add<std::vector<int>, int>
                (std::vector<int> const &vect, int const &init);
```

- To disambiguate among multiple possibilities detected by the compiler, explicit arguments may be used. For example:

```
std::vector<int> vi;
int sum = add<std::vector<int>, int>(vi, 0);
```

20.8 Specializing templates for deviating types

The initial `add()` template, defining two identically typed parameters works fine for all types sensibly supporting `operator+()` and a copy constructor. However, these assumptions are not always met. For example, when `char *`s are used, neither the `operator+()` nor the copy constructor is (sensibly) available. The compiler does not know this, and will try to instantiate the simple function template

```
template <typename Type>
Type add(Type const &t1, Type const &t2);
```

But it can't do so, since `operator+()` is not defined for pointers. In situations like these it is clear that a match between the template's type parameter(s) and the actually used type(s) is possible, but the standard implementation is pointless or produces errors.

To solve this problem a *template explicit specialization* may be defined. A template explicit specialization defines the function template for which a generic definition already exists using specific actual template type parameters.

In the abovementioned case an explicit specialization is required for a `char const *`, but probably also for a `char *` type. Probably, as the compiler still uses the standard type-deducing process mentioned earlier. So, when our `add()` function template is specialized for `char *` arguments, then its return type *must* also be a `char *`, whereas it *must* be a `char const *` if the arguments are `char const *` values. In these cases the template type parameter `Type` will be deduced properly. With `Type == char *`, for example, the head of the instantiated function becomes:

```
char *add(char *const &t1, char *const &t2)
```

If this is considered undesirable, an *overloaded* version could be designed expecting pointers. The following function template definition expects two (`const`) pointers, and returns a non-const pointer:

```
template <typename T>
T *add(T const *t1, T const *t2)
{
    std::cout << "Pointers\n";
    return new T;
}
```

But we might still not be where we want to be, as *this* overloaded version will now only accept pointers to constant `T` elements. Pointers to non-const `T` elements will not be accepted. At first sight it may come

as a surprise that the compiler will not apply a qualification transformation. But there's no need for the compiler to do so: when non-const pointers are used the compiler will simply use the initial definition of the `add()` template function expecting any two arguments of equal types.

So do we have to define yet another overloaded version, expecting non-const pointers? It is possible, but at some point it should become clear that our approach doesn't scale. Like ordinary functions and classes, templates should have well-described purposes. Trying to add overloaded template definitions to overloaded template definitions quickly turns the template into a kludge. Don't follow this approach. A better approach is probably to construct the template so that it fits its original purpose, make allowances for the occasional specific case, and to describe its purpose clearly in the template's documentation.

Nevertheless, there may be situations where a template explicit specialization may be worth considering. Two specializations for `const` and non-`const` pointers to characters might be considered for our `add()` function template. Template explicit specializations are constructed as follows:

- They start with the keyword `template`.
- Next, an empty set of angle brackets is written. This indicates to the compiler that there must be an *existing* template whose prototype matches the one we're about to define. If we err and there is no such template then the compiler reports an error like:

```
error: template-id 'add<char*>' for 'char* add(char* const&, char*
       const&)' does not match any template declaration
```

- Next the head of the function is defined, which must follow the same syntax as a template explicit instantiation declaration (see section 20.4.1): it must specify the correct returntype, function name, template type parameter explicitations, as well as the function's parameter list.
- The body of the function, defininig the special implementation that is required for the special actual template parameter types.

Here are two explicit specializations for the function template `add()`, expecting `char *` and `char const *` arguments (note that the `const` still appearing in the first template specialization is unrelated to the specialized type (`char *`), but refers to the `const &` mentioned in the original template's definition. So, in this case it's a reference to a constant pointer to a `char`, implying that the `char`s may be modified):

```
template <> char *add<char *>(char * const &p1,
                              char * const &p2)
{
    std::string str(p1);
    str += p2;
    return strcpy(new char[str.length() + 1], str.c_str());
}

template <> char const *add<char const *>(char const *const &p1,
                              char const *const &p2)
{
    static std::string str;
    str = p1;
    str += p2;
    return str.c_str();
}
```

Template explicit specializations are normally included in the file containing the other function template's implementations.

A template explicit specialization can be declared in the usual way. I.e., by replacing its body with a semicolon.

Note in particular how important the pair of angle brackets are that follow the `template` keyword when declaring a template explicit specialization. If the angle brackets were omitted, we would have constructed a template instantiation declaration. The compiler would silently process it, at the expense of a somewhat longer compilation time.

When declaring a template explicit specialization (or when using an instantiation declaration) the explicit specification of the template type parameters can be omitted if the compiler is able to deduce these types from the function's arguments. As this is the case with the `char (const) *` specializations, they could also be declared as follows:

```
template <> char *add(char * const &p1,
                                    char * const &p2)
template <> char const *add(char const *const &p1,
                                    char const *const &p2);
```

In addition, `template <>` could be omitted. However, this would remove the template character from the declaration, as the resulting declaration is now nothing but a plain function declaration. This is not an error: template functions and non-function templates may overload each other. Ordinary functions are not as restrictive as function templates with respect to allowed type conversions. This could be a reason to overload a template with an ordinary function every once in a while.

A function template explicit specialization is not just another overloaded version of the the function template. Whereas an overloaded version may define a completely different set of template parameters, a specialization must use the same set of template parameters as its non-specialized variant, but the compiler will use the specialization in situations where the actual template arguments match the types defined by the specialization. Subject to this restriction, many specializations can be defined.

20.9 SFINAE: Substitution Failure Is Not An Error

Consider the following struct definition:

```
struct Int
{
    typedef int type;
};
```

Although at this point it may seem strange to embed a typedef in a struct, as we'll see in chapter 22 there are certainly situations where this is actually very useful as it defines a standard interface to some core type defined in various template types. It allows us to define a variable of a kind that is required by the template. E.g., (the `typename` in the following function parameter list can be ignored for now. It's a subtlety that is explained in detail in section 22.1.1):

```
template <typename Type>
void func(typename Type::type value)
{
}
```

To call `func(10)` `Int` has to be specified explicitly since there may be many structs that define `type`: the compiler needs some assistance. The correct call is `func<Int>(10)`. Now it's clear that `Int` is meant and the compiler correctly deduces that `value` is an `int`.

But templates may be overloaded, and our next definition is:

```
template < typename Type >
void func(Type value)
{}
```

Now, to make sure this function is used we specify `func<int>(10)` and again this compiles flawlessly.

But as we've seen earlier in this chapter when the compiler determines which template to instantiate it creates a list of viable functions by matching the available function prototypes with the provided actual type. Therefore it has to determine the types of the parameters and herein lies a problem. When evaluating `Type = int` the compiler will encounter the prototypes `func(int::Type)` (first template definition) and `func(int)` (second template definition). But there is no `int::Type`, and so in a sense this generates an error. However, the error results from substituting the provided template type argument into the various template definitions. An type-problem caused by substituting a type in a template definition is *not* considered an error, but a mere indication that that particular type cannot be substituted in that particular template which is therefore removed from the list of candidate functions. This principle is known as *substitution failure is not an error* (SFINAE) and it is used often by the compiler to select not only a simple overloaded function (as shown here) but also to make the correct choice when selecting the appropriate specialization (see also chapters 21 and 22).

20.10 The function template selection mechanism

When the compiler encounters a function call, it must decide which function to call when overloaded functions are available. In this section this function selection mechanism is described.

In our discussion, we assume that we ask the compiler to compile the following `main()` function:

```
int main()
{
    double x = 12.5;
    add(x, 12.5);
}
```

Furthermore we assume that the compiler has seen the following six function declarations when it's about to compile `main()`:

```
template <typename Type>                                        // function 1
Type add(Type const &lvalue, Type const &rvalue);

template <typename Type1, typename Type2>                       // function 2
Type1 add(Type1 const &lvalue, Type2 const &rvalue);

template <typename Type1, typename Type2, typename Type3>   // function 3
Type1 add(Type1 const &lvalue, Type1 const &mvalue, Type2 const &rvalue);

double add(float lvalue, double rvalue);                        // function 4
double add(std::vector<double> const &vd);                      // function 5
double divide(double lvalue, double rvalue);                    // function 6
```

The compiler, having read `main()`'s statement, must now decide which function must actually be called. It proceeds as follows:

- First, a set of *candidate functions* is constructed. This set contains all functions that:
  - are visible at the point of the call;
  - have the same names as the called function.

  As function 6 has a different name, it is removed from the set. The compiler is left with a set of five candidate functions: 1 until 5.
- Second, the set of *viable functions* is constructed. Viable functions are functions for which type conversions exist that can be applied to match the types of the parameters of the functions and

the types of the actual arguments. This implies that the number of arguments must match the number of parameters of the viable functions.

- As functions 3 and 5 have different numbers of parameters they are removed from the set.
- Now let's 'play compiler' to decide among the remaining functions 1, 2 and 4. This is done by assigning *penalty points* to the remaining functions. Eventually the function having the smallest score will be selected. A point is assigned for every standard argument deduction process transformation that is required (so, for every *lvalue-*, *qualification-*, or *derived-to-base class* transformation that is applied).
- Eventually multiple functions might emerge at the top. Even though we have a draw in this case, the compiler will not always report an ambiguity. As we've seen before, a more specialized function is selected over a more general function. So, if a template explicit specialization and its more general variant appear at the top, the specialization is selected. Similarly, a ordinary function will be selected over a function template (but remember: only if both appear at the top of the ranking process).
- As a rule of thumb we have:
  - when there are multiple viable functions at the top of the set of viable functions, then the plain function template instantiations are removed;
  - if multiple functions remain, template explicit specializations are removed;
  - if only one function remains, it is selected;
  - otherwise, the compiler can't decide and reports an error: the call is ambiguous.

Now we apply the above procedure to the viable functions 1, 2 and 4. It happens that function 1 contains a slight complication, so we'll start with function 2.

- Function 2 has prototype:

  ```
  template <typename T1, typename T2>
  T1 add(T1 const &a, T2 const &b);
  ```

  The function is called as `add(x, 12.5)`. As `x` is a `double` both `T &x` and `T const &x` would be acceptable, albeit that `T const &x` will require a qualification transformation. Since the function's prototype uses `T const &` a qualification transformation is needed. The function is charged 1 point, and `T1` is now determined as `double`.

  Next, 12.5 is recognized as a `double` as well (note that `float` constants are recognized by their 'F' suffix, e.g., 12.5F), and it is also a constant value. So, without transformations, we find `12.5 == T2 const &` and at no charge `T2` is recognized as `double` as well.

- Function 4 has prototype:

  ```
  double add(float lvalue, double rvalue);
  ```

  Although it is called as `add(x, 12.5)` with `x` being of type `double`; but a standard conversion exists from type `double` to type `float`. Furthermore, 12.5 is a `double`, which can be used to initialize `rvalue`.

Thus, at this point we could ask the compiler to select among:

```
add(double const &, double const &b);
```

and

```
add(float, double);
```

This does not involve 'function template selection' since the first one has already been determined. As the first function doesn't require any standard conversion at all, it is selected, since a perfect match is selected over one requiring a standard conversion.

As an intermezzo you are invited to take a closer look at this process by defining `float x` instead of `double x`, or by defining `add(float x, double x)` as `add(double x, double x)`: in these cases the function template has the same prototype as the ordinary function, and so the ordinary function is selected since it's a more specific function. Earlier we've seen that process in action when redefining `ostream::operator»(ostream &os, string &str)` as an ordinary function.

Now it's time to go back to function template 1.

- Function 1 has prototype:

  ```
  template <typename T>
  T add(T const &t1, T const &t2);
  ```

  Once again we call `add(x, 12.5)` and will deduce template types. In this case there's only one template type parameter T. Let's start with the first parameter:

  - The argument `x` is of type `double`, so both `T &x` and `T const &x` are acceptable. According to the function's parameter list `T const &x` must be used, which requires a qualification transformation. So we'll charge the function 1 point and T is determined as `double`. This results in the instantiation of

    ```
    add(double const &t1, double const &t2)
    ```

    allowing us to call, at the expense of 1 point, `add(x, 12.5)`.

  But we can do better by starting our deduction process at the *second* parameter:

  - Since 12.5 is a constant `double` value we see that `12.5 == T const &`. So we conclude (free of charge) that T is `double`. Our function becomes

    ```
    add(double const &t1, double const &t2)
    ```

    allowing us to call `add(x, 12.5)`.

Earlier this section, we preferred function 2 over function 4. Function 2 is a function template that required one qualification transformation. Function 1, on the other hand, did not require any transformation at all, so it emerges as the function to be used.

As an exercise, feed the above six declarations and `main()` to the compiler and wait for the linker errors: the linker will complain that the (template) function

```
double add<double>(double const&, double const&)
```

is an undefined reference.

20.11 Compiling template definitions and instantiations

Consider the following definition of the `add()` function template:

```
template <typename Container, typename Type>
Type add(Container const &container, Type init)
{
    return std::accumulate(container.begin(), container.end(), init);
}
```

In this template definition, `std::accumulate()` is called using `container`'s `begin()` and `end()` members.

The calls `container.begin()` and `container.end()` are said to *depend on template type parameters*. The compiler, not having seen `container`'s interface, cannot check whether `container` will actually have members `begin()` and `end()` returning input iterators, as required by `std::accumulate`.

On the other hand, `std::accumulate()` itself is a function call which is independent of any template type parameter. Its *arguments* are dependent of template parameters, but the function call itself isn't. Statements in a template's body that are independent of template type parameters are said *not to depend on template type parameters*.

When the compiler reads a template definition, it will verify the syntactic correctness of all statements not depending on template type parameters. I.e., it must have seen all class definitions, all type definitions, all function declarations etc., that are used in the statements not depending on the template's type parameters. If this condition isn't met, the compiler will not accept the template's definition. Consequently, when defining the above template, the header file `numeric` must have been included first, as this header file declares `std::accumulate()`.

However, with statements depending on template type parameters the compiler cannot perform these extensive checks. E.g., it has no way to verify the existence of a member `begin()` for the as yet unspecified type `Container`. In these cases the compiler will perform superficial checks, assuming that the required members, operators and types will eventually become available.

The location in the program's source where the template is instantiated is called its *point of instantiation*. At the point of instantiation the compiler will deduce the actual types of the template's type parameters. At that point it will check the syntactic correctness of the template's statements that depend on template type parameters. This implies that *only at the point of instantiation* the required declarations must have been read by the compiler. As a rule of thumb, make sure that all required declarations (usually: header files) have been read by the compiler at every point of instantiation of the template. For the template's definition itself a more relaxed requirement can be formulated. When the definition is read only the declarations required for statements *not* depending on the template's type parameters must be known.

20.12 Static assertions (C++0x)

The `static_assert(constant expression, error message)` utility is defined by the C++0x standard to allow assertions to be made within template definitions. Here are two examples of its use:

```
static_assert(BUFSIZE1 != BUFSIZE2,
                            "BUFSIZE1 and BUFSIZE2 must be equal");

template <typename Type1, typename Type2>
void rawswap(Type1 &type1, Type2 &type2)
{
    static_assert(sizeof(Type1) != sizeof(Type2),
                    "rawswap: Type1 and Type2 must have equal sizes");
    // ...
}
```

The first example shows how to avoid yet another preprocessor directive. In this case the `#error` directive; the second example shows how `static_asseert` can be used to ensure that a template operates under the right condition(s).

Note that the `static_assert` is a compile-time matter and that it doesn't have any effect on the run-time efficiency of the code in which it is used.

20.13 Summary of the template declaration syntax

In this section the basic syntactic constructions when declaring templates are summarized. When *defining* templates, the terminating semicolon should be replaced by a function body. However, not every template declaration may be converted into a template definition. If a definition may be provided it is explicitly mentioned.

- A plain template declaration (a definition may be provided instead of the declaration's semicolon):

```
template <typename Type1, typename Type2>
void function(Type1 const &t1, Type2 const &t2);
```

- A template instantiation declaration (no definition, no template parameter list following `template`):

```
template
void function<int, double>(int const &t1, double const &t2);
```

- A template using explicit types (no definition):

```
void (*fp)(double, double) = function<double, double>;
void (*fp)(int, int) = function<int, int>;
```

- A template specialization (an empty template parameter list, a definition may be provided instead of the declaration's semicolon):

```
template <>
void function<char *, char *>(char *const &t1, char *const &t2);
```

- A template declaration declaring friend function templates within class templates (covered in section 21.9):

```
friend void function<Type1, Type2>(parameters);
```

Chapter 21

Class Templates

Templates can not only be constructed for functions but also for complete classes. Constructing a class template can be considered when the class should be able to handle different types of data. Class templates are frequently used in **C++**: chapter 12 discusses data structures like `vector, stack` and `queue`, which are implemented as *class templates*. With class templates, the algorithms and the data on which the algorithms operate are completely separated from each other. To use a particular data structure, operating on a particular data type, only the data type needs to be specified when the class template object is defined or declared, e.g., `stack<int> iStack`.

In this chapter constructing and using class templates is discussed. In a sense, class templates compete with object oriented programming (cf. chapter 14), where a mechanism somewhat similar to templates is seen. Polymorphism allows the programmer to postpone the definitions of algorithms, by deriving classes from a base class in which the algorithm is only partially implemented, while the data upon which the algorithms operate may first be defined in derived classes, together with member functions that were defined as pure virtual functions in the base class to handle the data. On the other hand, templates allow the programmer to postpone the specification of the data upon which the algorithms operate. This is most clearly seen with the abstract containers, completely specifying the algorithms but at the same time leaving the data type on which the algorithms operate completely unspecified.

The correspondence between class templates and polymorphic classes is well-known. In their book **C++ Coding Standards** (Addison-Wesley, 2005) Sutter and Alexandrescu (2005) refer to *static polymorphism* and *dynamic polymorphism*. *Dynamic* polymorphism is what we use when overriding virtual members: Using the *vtable* construction the function that's actually called depends on the type of object a (base) class pointer points to. *Static* polymorphism is used when templates are used: depending on the actual types, the compiler *creates* the code, compile time, that's appropriate for those particular types. There's no need to consider static and dynamic polymorphism as mutually exlusive variants of polymorphism. Rather, both can be used together, combining their strengths. A warning is in place, though. When a class template defines virtual members *all* virtual members are instantiated for every instantiated type. This has to happen, since the compiler must be able to construct the class's `vtable`.

Generally, class templates are easier to use. It is certainly easier to write `stack<int> istack` to create a stack of `int`s than to derive a new class `Istack: public stack` and to implement all necessary member functions to be able to create a similar stack of `int`s using object oriented programming. On the other hand, for each different type that is used with a class template the complete class is reinstantiated, whereas in the context of object oriented programming the derived classes *use*, rather than *copy*, the functions that are already available in the base class (but see also section 21.10).

21.1 Defining class templates

Now that we've covered the construction of function templates, we're ready for the next step: constructing class templates. Many useful class templates already exist. Instead of illustrating how an existing class template was constructed, let's discuss the construction of a useful new class template.

In chapter 18 we've encountered the `auto_ptr` class (section 18.3.7). The `auto_ptr` (in the C++0x standard deprecated in favor of `unique_ptr` and `shared_ptr` (cf. sections 18.3 and 18.4)) allows an object to act like a pointer.

One of the disadvantages of `auto_ptr` is that it can only be used for single objects and not for pointers to arrays of objects. This can be remedied, and here we'll construct the class template `FBB::auto_ptr`, behaving like `auto_ptr`, but managing a pointer to an array of objects. It must be stressed that `FBB::auto_ptr` is only developed here as an illustration. The STL classes `unique_ptr` and `shared_ptr` should be used in 'real life'.

Using an existing class as our point of departure shows an important design principle: it's often easier to construct a template (function or class) from an existing template than to construct the template completely from scratch. In this case the existing `std::auto_ptr` acts as our model. Therefore, we want to provide the class with the following members:

- Constructors to create an object of the class `FBB::auto_ptr`;
- A destructor;
- An overloaded `operator=()`;
- An `operator[]()` to retrieve and reassign the elements given their indices.
- All other members of `std::auto_ptr`, with the exception of the dereference operator (`operator*()`), since our `FBB::auto_ptr` object will hold multiple objects, and although it would be entirely possible to define it as a member returning a reference to the first element of its array of objects, the member `operator+(int index)`, returning the address of object `index` would most likely be expected too. These extensions of `FBB::auto_ptr` are left as exercises to the reader.

Now that we have decided which members we need, the class interface can be constructed. Like function templates, a class template definition begins with the keyword `template`, which is also followed by a non-empty list of template type and/or non-type parameters, surrounded by angle brackets. The `template` keyword followed by the template parameter list enclosed in angle brackets is called a *template announcement* in the **C++** Annotations. In some cases the template announcement's parameter list may be empty, leaving only the angle brackets.

Following the template announcement the class interface is provided, in which the formal template type parameter names may be used to represent types and constants. The class interface is constructed as usual. It starts with the keyword `class` and ends with a semicolon.

Normal design considerations should be followed when constructing class template member functions or class template constructors: class template type parameters should preferably be defined as `Type const &`, rather than `Type`, to prevent unnecessary copying of large data structures. Template class constructors should use member initializers rather than member assignment within the body of the constructors, again to prevent double assignment of composed objects: once by the default constructor of the object, once by the assignment itself.

Here is our initial version of the class `FBB::auto_ptr` showing all its members:

```
namespace FBB
{
    template <typename Data>
    class auto_ptr
    {
        Data *d_data;

        public:
            auto_ptr();
            auto_ptr(auto_ptr<Data> &other);
            auto_ptr(Data *data);
            ~auto_ptr();
```

```
        auto_ptr<Data> &operator=(auto_ptr<Data> &rvalue);
        Data &operator[](size_t index);
        Data const &operator[](size_t index) const;
        Data *get();
        Data const *get() const;
        Data *release();
        void reset(Data *p = 0);
    private:
        void destroy();
        void copy(auto_ptr<Data> &other);
        Data &element(size_t idx) const;
};

template <typename Data>
inline auto_ptr<Data>::auto_ptr()
:
    d_data(0)
{}

template <typename Data>
inline auto_ptr<Data>::auto_ptr(auto_ptr<Data> &other)
{
    copy(other);
}

template <typename Data>
inline auto_ptr<Data>::auto_ptr(Data *data)
:
    d_data(data)
{}

template <typename Data>
inline auto_ptr<Data>::~auto_ptr()
{
    destroy();
}

template <typename Data>
inline Data &auto_ptr<Data>::operator[](size_t index)
{
    return d_data[index];
}

template <typename Data>
inline Data const &auto_ptr<Data>::operator[](size_t index) const
{
    return d_data[index];
}

template <typename Data>
inline Data *auto_ptr<Data>::get()
{
    return d_data;
}

template <typename Data>
inline Data const *auto_ptr<Data>::get() const
{
    return d_data;
```

```
    }

    template <typename Data>
    inline void auto_ptr<Data>::destroy()
    {
        delete[] d_data;
    }

    template <typename Data>
    inline void auto_ptr<Data>::copy(auto_ptr<Data> &other)
    {
        d_data = other.release();
    }

    template <typename Data>
    auto_ptr<Data> &auto_ptr<Data>::operator=(auto_ptr<Data> &rvalue)
    {
        if (this != &rvalue)
        {
            destroy();
            copy(rvalue);
        }
        return *this;
    }

    template <typename Data>
    Data *auto_ptr<Data>::release()
    {
        Data *ret = d_data;
        d_data = 0;
        return ret;
    }

    template <typename Data>
    void auto_ptr<Data>::reset(Data *ptr)
    {
        destroy();
        d_data = ptr;
    }

} // FBB
```

The class interface shows the following features:

- If it is assumed that the template type `Data` is an ordinary type, the class interface appears to have no special characteristics at all. It looks like any old class interface. This is generally true. Often a template class can easily be constructed after having constructed the class for one or two ordinary types, followed by an abstraction phase changing all necessary references to ordinary data types into generic data types, which then become the template's type parameters.
- At closer inspection, some special characteristics can actually be discerned. The parameters of the class's copy constructor and overloaded assignment operators aren't references to plain `auto_ptr` objects, but rather references to `auto_ptr<Data>` objects. Class template objects (or their references or pointers) *always* require the template type parameters to be specified.
- Different from the standard design of copy constructors and overloaded assignment operators, their parameters are *non*-`const` references. This has nothing to do with the class being a class template, but is a consequence of `auto_ptr`'s design itself: both the copy constructor and the overloaded assignment operator take the other's object's pointer, effectively changing the other object into a 0-pointer.

- Like ordinary classes, members can be defined *inline*. Actually, *all* class template members are defined inline (when using precompiled templates *precompiled templates* this doesn't change; it only means that the compiler has reorganized the template definition so that it can process the definition faster). As noted in section 7.6, the definition may be put inside the class interface or outside (i.e., following) the class interface. As a rule of thumb the same design principles should be followed here as with ordinary classes: they should be defined below the interface to keep the interface clean and readable. Long implementations in the interface tend to obscure the interface itself.
- When objects of a class template are instantiated, the definitions of all the template's member functions that are used (but *only* those) must have been seen by the compiler. Although that characteristic of templates could be refined to the point where each definition is stored in a separate function template definition file, including only the definitions of the function templates that are actually needed, it is hardly ever done that way (even though it would speed up the required compilation time). Instead, the usual way to define class templates is to define the interface, defining some functions inline, and to define the remaining function templates immediately below the class template's interface.
- Beside the dereference operator (`operator*()`), the well-known pair of `operator[]()` members are defined. Since the class receives no information about the size of the array of objects, these members cannot support array-bound checking.

Let's have a look at some of the member functions defined beyond the class interface. Note in particular:

- The definition below the interface is the actual template definition. Since it is a definition it must start with a `template phrase`. The function's *declaration* must also start with a `template` phrase, but that is implied by the interface itself, which already provides the required phrase at its very beginning;
- Wherever `auto_ptr` is mentioned in the implementation, the template's type parameter is mentioned as well. This is obligatory. Actually, the class template's type name is the name of the class template plus its template argument. Thus, a `vector<int>` represents another *class type* than a `vector<float>`.

Some remarks about specific members:

- The advised `copy()` and `destroy()` members (see section 8.5.1) are very simple, but were added to the implementation to promote standardization of classes containing pointer members.
- The overloaded assignment constructor still has to check for auto-assignment.

Now that the class has been defined, it can be used. To use the class, its object must be instantiated for a particular data type. The example defines a new `std::string` array, storing all command-line arguments. Then, the first command-line argument is printed. Next, the `auto_ptr` object is used to initialize another `auto_ptr` of the same type. It is shown that the original `auto_ptr` now holds a 0-pointer, and that the second `auto_ptr` object now holds the command-line arguments:

```
#include <iostream>
#include <algorithm>
#include <string>
#include "autoptr.h"
using namespace std;

int main(int argc, char **argv)
{
    FBB::auto_ptr<string> sp(new string[argc]);
    copy(argv, argv + argc, sp.get());

    cout << "First auto_ptr, program name: " << sp[0] << endl;
```

```
    FBB::auto_ptr<string> second(sp);

    cout << "First auto_ptr, pointer now: " << sp.get() << endl;
    cout  << "Second auto_ptr, program name: " << second[0] << endl;

    return 0;
}
/*
    Generated output:

    First auto_ptr, program name: a.out
    First auto_ptr, pointer now: 0
    Second auto_ptr, program name: a.out
*/
```

21.1.1 Default class template parameters

Different from function templates, template parameters of template classes may be given default values. This holds true both for template type- and template non-type parameters. If a class template is instantiated without specifying arguments for its template parameters, and if default template parameter values were defined, then the defaults are used. When defining such defaults keep in mind that the defaults should be suitable for the majority of instantiations of the class. E.g., for the class template `FBB::auto_ptr` the template's type parameter list could have been altered by specifying `int` as its default type:

```
template <typename Data = int>
```

Even though default arguments can be specified, the compiler must still be informed that object definitions refer to templates. So, when instantiating class template objects for which default parameter values have been defined the type specifications may be omitted, but the angle brackets must remain. So, assuming a default type for the `FBB::auto_ptr` class, an object of that class may be defined as:

```
FBB::auto_ptr<> intAutoPtr;
```

No defaults must be specified for template members defined outside of their class interface. Function templates, even member function templates, cannot specify default parameter values. So, the definition of, e.g., the `release()` member will always begin with the same `template` specification:

```
template <typename Data>
```

When a class template uses multiple template parameters, all may be given default values. However, like default function arguments, once a default value is used, all remaining parameters must also use their default values. A template type specification list may not start with a comma, nor may it contain multiple consecutive commas.

21.1.2 Declaring class templates

Class templates may also be *declared*. This may be useful in situations where forward class declarations are required. To declare a class template, simply remove its interface (the part between the curly braces):

```
namespace FBB
{
```

```
        template <typename Type>
        class auto_ptr;
    }
```

Here default types may also be specified. However, default type values cannot be specified in both the declaration and the definition of a template class. As a rule of thumb default values should be omitted from *declarations*, as class template declarations are never used when instantiating objects, but only for the occasional forward reference. Note that this differs from default parameter value specifications for member functions in ordinary classes. Such defaults should be specified in the member functions' declarations and *not* in their definitions.

21.1.3 Preventing template instantiations (C++0x)

In **C++** templates are instantiated when their address is taken or when they are used. As described in section 21.1.2 it is possible to forward declare a class template in order to allow the definition of a pointer or reference to that template class or to allow it being used as a return type.

In other situations templates are instantiated when they are being used. If this happens many times (i.e., in many different source files) then this may slow down the compilation process considerably. The C++0x standard allows programmers to *prevent* templates from being instantiated. For this the `extern template` syntax is introduced. The following example declares the `std::vector<int>` template:

```
    extern template class std::vector<int>;
```

Having declared the class template it can be used in its translation unit. E.g., the following function will properly compile:

```
#include <vector>
#include <iostream>
using namespace std;

extern template class vector<int>;

void vectorUser()
{
    vector<int> vi;
    cout << vi.size() << endl;
}
```

Note, however:

- The declaration by itself does not make the class definition available. The `vector` header file still needs to be included to make the features of the class vector known to the compiler;
- The compiler *assumes* (as it always does) that what's declared here is implemented elsewhere. But in this case the compiler encounters an *implicit declaration*: the features of the vector class that are actually used by the above program are not individually declared but they are declared as a group, using the `extern template` declaration. This not only holds true for explicitly used members, but any hidden members (copy constructors, move constructors, conversion operators, constructors called during promotions, to name a few): all are assumed by the compiler to have been instantiated elsewhere;
- The above source file will *compile* but the compiler must find the instantiations before its linker can produce a final program. To accomplish this one or more sourcefiles may be constructed in which all the required instantiations are eventually made available.

In a stand-alone program one might postpone defining the required members and wait for the linker to complain about unresolved external references. These may then be used to create a series of instantiation declarations which are then linked with the program to satisfy the linker. Not a very simple task, though, as the declarations must strictly match the way the members are declared in the class interface. An easier approach is to define an *instantiation source file* in which all the facilities that are used by the program are actually instantiated in a function that is not called by the program. By adding this instantiation function to the source file containing `main` we can be sure that all required members will be instantiated as well. Here is an example of how this may be done:

```
#include <vector>
#include <iostream>

extern void vectorUser();

int main()
{
    vectorUser();
}

// this part is never called. It is added to make sure all required
// features of declared templates will also be instantiated.

namespace
{
    void instantiator()
    {
        std::vector<int> vi;
        vi.size();
    }
}
```

21.1.4 Non-type parameters

As we've seen with function templates, template parameters are either template type parameters or template non-type parameters (actually, a third kind of template parameter exists, the *template template parameter*, which is discussed in chapter 22 (section 22.3)).

Class templates also may define non-type parameters. Like the non-const parameters used with function templates they must be constants whose values are known by the time an object is instantiated.

However, their values are not deduced by the compiler using arguments passed to constructors. Assume we modify the class template `FBB::auto_ptr` so that it has an additional non-type parameter `size_t Size`. Next we use this `Size` parameter in a new constructor defining an array of `Size` elements of type `Data` as its parameter. The new `FBB::auto_ptr` template class becomes (showing only the relevant constructors (the reason for using the type `Data2` is given in section 21.1.5); note the two template type parameters that are now required, e.g., when specifying the type of the copy constructor's parameter):

```
namespace FBB
{
    template <typename Data, size_t Size>
    class auto_ptr
    {
        Data *d_data;
        size_t d_n;

        public:
            auto_ptr(auto_ptr<Data, Size> &other);
```

```
            auto_ptr(Data2 *data);
            auto_ptr(Data const (&arr)[Size]);
            ...
    };

    template <typename Data, size_t Size>
    inline auto_ptr<Data, Size>::auto_ptr(Data const (&arr)[Size])
    :
        d_data(new Data2[Size]),
        d_n(Size)
    {
        std::copy(arr, arr + Size, d_data);
    }
}
```

Unfortunately, this new setup doesn't satisfy our needs, as the values of template non-type parameters are not deduced by the compiler. When the compiler is asked to compile the following `main()` function it reports a mismatch between the required and actual number of template parameters:

```
int main()
{
    int arr[30];

    FBB::auto_ptr<int> ap(arr);
}
/*
    Error reported by the compiler:

    In function ‘int main()’:
        error: wrong number of template arguments (1, should be 2)
        error: provided for ‘template<class Data, size_t Size>
               class FBB::auto_ptr’
*/
```

Defining `Size` as a non-type parameter having a default value doesn't work either. The compiler will use the default, unless explicitly specified otherwise. So, reasoning that `Size` can be 0 unless we need another value, we might specify `size_t Size = 0` in the templates parameter type list. However, this causes a mismatch between the default value 0 and the actual size of the array `arr` as defined in the above `main()` function. The compiler using the default value, reports:

```
In instantiation of ‘FBB::auto_ptr<int, 0>’:
...
error: creating array with size zero (‘0’)
```

So, although class templates may use non-type parameters, they must be specified like the type parameters when an object of the class is defined. Default values can be specified for those non-type parameters, but then the default will be used when the non-type parameter is left unspecified.

Note that *default template parameter values* (either type or non-type template parameters) may *not* be used when template member functions are defined outside the class interface. Function template definitions (and thus: class template member functions) may not be given default template (non) type parameter values. If default template parameter values are to be used for class template members, they have to be specified in the class interface.

Similar to non-type parameters of function templates, non-type parameters of class templates may only be specified as constants:

- Global variables have constant addresses, which can be used as arguments for non-type parameters.

- Local and dynamically allocated variables have addresses that are not known by the compiler when the source file is compiled. These addresses can therefore not be used as arguments for non-type parameters.
- Lvalue transformations are allowed: if a pointer is defined as a non-type parameter, an array name may be specified.
- Qualification conversions are allowed: a pointer to a non-const object may be used with a non-type parameter defined as a `const` pointer.
- Promotions are allowed: a constant of a 'narrower' data type may be used for the specification of a non-type parameter of a 'wider' type (e.g., a `short` can be used when an `int` is called for, a `long` when a `double` is called for).
- Integral conversions are allowed: if an `size_t` parameter is specified, an `int` may be used too.
- Variables cannot be used to specify template non-type parameters, as their values are not constant expressions. Variables defined using the `const` modifier, however, may be used, as their values never change.

Although our attempts to define a constructor of the class `FBB::auto_ptr` accepting an array as its argument, allowing us to use the array's size within the constructor's code has failed so far, we're not yet out of options. In the next section an approach will be described allowing us to reach our goal, after all.

21.1.5 Member templates

Our previous attempt to define a template non-type parameter which is initialized by the compiler to the number of elements of an array failed because the template's parameters are not implicitly deduced when a constructor is called, but they are explicitly specified, when an object of the class template is defined. As the parameters are specified just before the template's constructor is called, there's nothing to deduce anymore, and the compiler will simply use the explicitly specified template arguments.

On the other hand, when template *functions* are used, the actual template parameters are deduced from the arguments used when calling the function. This opens an approach route to the solution of our problem. If the constructor itself is made into a member which itself is a function template (containing a template announcement of its own), then the compiler will be able to deduce the non-type parameter's value, without us having to specify it explicitly as a class template non-type parameter.

Member functions (or classes) of class templates which themselves are templates are called *member templates*. Member templates are defined in the same way as any other template, including the `template <typename ...>` header.

When converting our earlier `FBB::auto_ptr(Data const (&array)[Size])` constructor into a member template we may use the class template's `Data` type parameter, but must provide the member template with a non-type parameter of its own. The class interface is given the following additional member declaration:

```
template <typename Data>
class auto_ptr
{
    ...
    public:
        template <size_t Size>
        auto_ptr(Data const (&arr)[Size]);
    ...
};
```

and the constructor's implementation becomes:

```
template <typename Data>
template <size_t Size>
inline auto_ptr<Data>::auto_ptr(Data const (&arr)[Size])
:
    d_data(new Data[Size]),
    d_n(Size)
{
    std::copy(arr, arr + Size, d_data);
}
```

Member templates have the following characteristics:

- Normal access rules apply: the constructor can be used by the general program to construct an `FBB::auto_ptr` object of a given data type. As usual for class templates, the data type must be specified when the object is constructed. To construct an `FBB::auto_ptr` object from the array `int array[30]` we define:

  ```
  FBB::auto_ptr<int> object(array);
  ```

- Any member can be defined as a member template, not just a constructor.
- When a template member is defined below its class, the class template parameter list must precede the function template parameter list of the template member. Furthermore:
  - The member should be defined inside its proper namespace environment. The organization within files defining class templates within a namespace should therefore be:

    ```
    namespace SomeName
    {
        template <typename Type, ...>   // class template definition
        class ClassName
        {
            ...
        };

        template <typename Type, ...>   // non-inline member definition(s)
        ClassName<Type, ...>::member(...)
        {
            ...
        }
    }                                   // namespace closed
    ```

  - Two `template` announcements must be used: the class template's `template` announcement is specified first, followed by the member template's `template` announcement.
  - The definition itself must specify the member template's proper scope: the member template is defined as a member of the class `FBB::auto_ptr`, instantiated for the formal template parameter type `Data`. Since we're already inside the namespace `FBB`, the function header starts with `auto_ptr<Data>::auto_ptr`.
  - The formal template parameter names in the declaration and implementation must be identical.

One small problem remains. When we're constructing an `FBB::auto_ptr` object from a fixed-size array the above constructor is not used. Instead, the constructor `FBB::auto_ptr<Data>::auto_ptr(Data *data)` is activated. As the latter constructor is not a member template, it is considered a more specialized version of a constructor of the class `FBB::auto_ptr` than the former constructor. Since both constructors accept an array the compiler will call `auto_ptr(Data *)` rather than `auto_ptr(Data const (&array)[Size])`. This problem can be solved by simply changing the constructor `auto_ptr(Data *data)` into a member template as well, in which case its template type parameter should be changed into '`Data`'. The only remaining subtlety is that template parameters of member templates may not

shadow the template parameters of their class. Renaming `Data` into `Data2` takes care of this subtlety. Here is the (inline) definition of the `auto_ptr(Data *)` constructor, followed by an example in which both constructors are actually used:

```
template <typename Data>
template <typename Data2>           // data: dynamically allocated
inline auto_ptr<Data>::auto_ptr(Data2 *data)
:
    d_data(data),
    d_n(0)
{}
```

Calling both constructors in `main()`:

```
int main()
{
    int array[30];

    FBB::auto_ptr<int> ap(array);
    FBB::auto_ptr<int> ap2(new int[30]);

    return 0;
}
```

21.1.6 Computing the return type of function objects (C++0x)

As amply illustrated in chapter 19 function objects play an important role when using generic algorithms. Like generic algorithms themselves, function objects can be generically defined as members of class templates. When the function call operator (`operator()()`) of such classes themselves have arguments, then the types of those arguments may be abstracted as well by defining the function call operator as a member template. Consider the following example:

```
template <typename Class>
class Filter
{
    Class obj;
    public:
        template <typename Arg>
        Arg operator()(Arg const &arg) const
        {
            return obj(arg);
        }
};
```

The `Filter` class template is a wrapper around `Class`, filtering `Class`'s overloaded function call through its own function call operator. In the above example the return value of `Class`'s function call operator is simply passed on, but any other manipulation is possible as well.

A particular type specified with `Filter`'s instantiation may have multiple function call operators, e.g.:

```
struct Math
{
    int operator()(int x);
    double operator()(double x);
};
```

Math objects can now be filtered defining Filter<Math> fm and then either using Math's first or second function call operator, depending on the actual argument type. With fm(5) the int-version is used, with fm(12.5) the double-version is used.

Unfortunately the scheme doesn't work if the function call operators have different return and argument types. E.g., the following class Convert cannot be used with Filter:

```
struct Convert
{
    double operator()(int x);
    std::string operator()(double x);
};
```

This problem can be tackled successfully using defines the class template std::result_of<Functor(Typelist)> defined by the C++0x standard. This class template defines type representing the type returned by Functor<TypeList>. It can be used to improve Filter as follows:

```
template <typename Class>
class Filter
{
    Class obj;
    public:
        template <typename Arg>
        typename std::result_of<Class(Arg)>::type
        Arg operator()(Arg const &arg) const
        {
            return obj(arg);
        }
};
```

Using this definition, Filter<Convert> fc can be constructed and fc(5) will return a double, while fc(4.5) returns a std::string.

The template type parameter that is used when result_of is used must represent the type(s) of a function call operator. It starts with the operator's class type (which may be a template type parameter, as shown), followed by a (possibly empty) list of types. The compiler will match the provided specification with existing function call operators and will define the template's ::type as the return type of the matched function call operator.

To use the std::result_of template the header file functional must be included.

21.2 Static data members

When static members are defined in class templates, they are instantiated for every new instantiation. As they are static members, there will be only one member when multiple objects of the *same* template type(s) are defined. For example, in a class like:

```
template <typename Type>
class TheClass
{
    static int s_objectCounter;
};
```

There will be *one* TheClass<Type>::objectCounter for each different Type specification. The following instantiates just one single static variable, shared among the different objects:

```
TheClass<int> theClassOne;
TheClass<int> theClassTwo;
```

Mentioning static members in interfaces does not mean these members are actually defined: they are only *declared* by their classes and must be *defined* separately. With static members of class templates this is not different. The definitions of static members are usually provided immediately following (i.e., below) the template class interface. The static member `s_objectCounter` will thus be defined as follows, just below its class interface:

```
template <typename Type>                    // definition, following
int TheClass<Type>::s_objectCounter = 0;    // the interface
```

In the above case, `s_objectCounter` is an `int` and thus independent of the template type parameter `Type`.

In a list-like construction, where a pointer to objects of the class itself is required, the template type parameter `Type` must be used to define the static variable, as shown in the following example:

```
template <typename Type>
class TheClass
{
    static TheClass *s_objectPtr;
};

template <typename Type>
TheClass<Type> *TheClass<Type>::s_objectPtr = 0;
```

As usual, the definition can be read from the variable name back to the beginning of the definition: `s_objectPtr` of the class `TheClass<Type>` is a pointer to an object of `TheClass<Type>`.

Finally, when a static variable of a template's type parameter is defined, it should of course not be given the initial value 0. The default constructor (e.g., `Type()` will usually be more appropriate):

```
template <typename Type>                    // s_type's definition
Type TheClass<Type>::s_type = Type();
```

21.3 Specializing class templates for deviating types

Our earlier class `FBB::auto_ptr` can be used for many different types. Their common characteristic is that they can simply be assigned to the class's `d_data` member, e.g., using `auto_ptr(Data *data)`. However, this is not always as simple as it looks. What if `Data`'s actual type is `char *`? Examples of a `char **`, `data`'s resulting type, are well-known: `main()`'s `argv` parameter and and `environ`, for example are `char **` varables, both having a final element equal to 0. The specialization developed here assumes that there is indeed such a final 0-pointer element.

It this special case we might not be interested in the mere reassignment of the constructor's parameter to the class's `d_data` member, but we might be interested in copying the complete `char **` structure. To implement this, class template specializations may be used.

Class template specializations are used in cases where member function templates cannot (or should not) be used for a particular actual template parameter type. In those cases specialized template members can be constructed, fitting the special needs of the actual type.

Class template member specializations are specializations of existing class members. Since the class members already exist, the specializations will *not* be part of the class interface. Rather, they are defined below the interface as members, redefining the more generic members using explicit types. Furthermore, as they are specializations of existing class members, their function prototypes must

exactly match the prototypes of the member functions for which they are specializations. For our `Data = char *` specialization the following definition could be designed:

```
template <>
auto_ptr<char *>::auto_ptr(char **argv)
:
    d_n(0)
{
    char **tmp = argv;
    while (*tmp++)
        d_n++;
    d_data = new char *[d_n];

    for (size_t idx = 0; idx < d_n; idx++)
    {
        std::string str(argv[idx]);
        d_data[idx] =
            strcpy(new char[str.length() + 1], str.c_str());
    }
}
```

Now, the above specialization will be used to construct the following `FBB::auto_ptr` object:

```
int main(int argc, char **argv)
{
    FBB::auto_ptr<char *> ap3(argv);
    return 0;
}
```

Although defining a template member specialization may allow us to use the occasional exceptional type, it is also quite possible that a single template member specialization is not enough. Actually, this is the case when designing the `char *` specialization, since the template's `destroy()` implementation is not correct for the specialized type `Data = char *`. When multiple members must be specialized for a particular type, then a complete class template specialization might be considered.

A completely specialized class shows the following characteristics:

- The class template specialization follows the generic class template definition. After all, it's a specialization, so the compiler must have seen what is being specialized.
- All the class's template parameters are given specific type names or (for the non-type parameters) specific values. These specific values are explicitly stated in a template parameter specification list (surrounded by angle brackets) which is inserted immediately following the template's class name.
- All the specialized template members specify the specialized types and values where the generic template parameters are used in the generic template definition.
- Not all the template's members *have* to be defined, but, to ensure generality of the specialization, *should* be defined. If a member is left out of the specialization, it can't be used for the specialized type(s).
- Additional members may be defined in the specialization. However, those that are defined in the generic template too must have corresponding members (using the same prototypes, albeit using the generic template parameters) in the generic class template definition. The compiler will not complain when additional members are defined, and will allow you to use those members with objects of the specialized class template.
- Member functions of specialized class templates may be defined within their specializing class or they may be declared in the specializing class. When they are only declared, then their definitition

should be given below the specialized class template's interface. Such an implementation may *not* begin with a `template <>` announcement, but should immediately start with the member function's header.

Below a full specialization of the class template `FBB::auto_ptr` for the actual type `Data = char *` is given, illustrating the above characteristics. The specialization should be appended to the file already containing the generic class template. To reduce the size of the example members that are only declared may be assumed to have identical implementations as used in the generic template.

```
#include <iostream>
#include <algorithm>
#include "autoptr.h"

namespace FBB
{
    template<>
    class auto_ptr<char *>
    {
        char **d_data;
        size_t d_n;

        public:
            auto_ptr<char *>();
            auto_ptr<char *>(auto_ptr<char *> &other);
            auto_ptr<char *>(char **argv);

            // template <size_t Size>             NI
            // auto_ptr(char *const (&arr)[Size])

            ~auto_ptr();
            auto_ptr<char *> &operator=(auto_ptr<char *> &rvalue);
            char *&operator[](size_t index);
            char *const &operator[](size_t index) const;
            char **get();
            char *const *get() const;
            char **release();
            void reset(char **argv);
            void additional() const;    // just an additional public
                                        // member
        private:
            void full_copy(char **argv);
            void copy(auto_ptr<char *> &other);
            void destroy();
    };

    inline auto_ptr<char *>::auto_ptr()
    :
        d_data(0),
        d_n(0)
    {}

    inline auto_ptr<char *>::auto_ptr(auto_ptr<char *> &other)
    {
        copy(other);
    }

    inline auto_ptr<char *>::auto_ptr(char **argv)
    {
        full_copy(argv);
```

```
    }

    inline auto_ptr<char *>::~auto_ptr()
    {
        destroy();
    }

    inline void auto_ptr<char *>::reset(char **argv)
    {
        destroy();
        full_copy(argv);
    }

    inline void auto_ptr<char *>::additional() const
    {}

    inline void auto_ptr<char *>::full_copy(char **argv)
    {
        d_n = 0;
        char **tmp = argv;
        while (*tmp++)
            d_n++;
        d_data = new char *[d_n];

        for (size_t idx = 0; idx < d_n; idx++)
        {
            std::string str(argv[idx]);
            d_data[idx] =
                strcpy(new char[str.length() + 1], str.c_str());
        }
    }

    inline void auto_ptr<char *>::destroy()
    {
        while (d_n--)
            delete d_data[d_n];
        delete[] d_data;
    }
}
```

Note that specializations not automatically have empty template parameter lists. Consider the following example of an (grossly incomplete) specialization of `FBB::auto_ptr`:

```
#include <iostream>
#include <vector>
#include "autoptr.h"

namespace FBB
{
    template<typename T>
    class auto_ptr<std::vector<T> *>
    {
        public:
            auto_ptr<std::vector<T> *>();
    };

    template <typename T>
    inline auto_ptr<std::vector<T> * >::auto_ptr()
    {
```

```
            std::cout << "Vector specialization\n";
        }
    }
```

In this example a specialization is created for the type `std::vector`, instantiated with any data type `T`. Since `T` is not specified, it must be mentioned in the template paramter list as a template type parameter. E.g., if an `FBB::auto_ptr<std::vector<int> *>` is constructed, the compiler deducts that `T` is an `int` and will use the `vector<T> *` specialization, in which `T` could be used as a type specification. The following basic example shows that the compiler will indeed select the `vector<T>*` specialization:

```
#include "autoptr4.h"

int main(int argc, char **argv)
{
    FBB::auto_ptr<std::vector<int> *> vspec;
    return 0;
}
```

21.4 Partial specializations

In the previous section we've seen that it is possible to design template class specializations. It was shown that both class template members and complete class templates could be specialized. Furthermore, the specializations we've seen were specializing template type parameters.

In this section we'll introduce a variant of these specializations, both in number and types of template parameters that are specialized. *Partial specializations* may be defined for class templates having multiple template parameters. With partial specializations a subset (any subset) of template type parameters are given specific values.

Having discussed specializations of template type parameters in the previous section, we'll discuss specializations of non-type parameters in the current section. Partial specializations of template non-type parameters will be illustrated using some simple concepts defined in matrix algebra, a branch of linear algebra.

A matrix is commonly thought of as consisting of a table of a certain number of rows and columns, filled with numbers. Immediately we recognize an opening for using templates: the numbers might be plain `double` values, but they could also be complex numbers, for which our *complex container* (cf. section 12.4) might prove useful. Consequently, our class template should be given a `DataType` template type parameter, for which a ordinary class can be specified when a matrix is constructed. Some simple matrices using `double` values, are:

```
1   0   0           An identity matrix,
0   1   0           a 3 x 3 matrix.
0   0   1

1.2  0    0    0    A rectangular matrix,
0.5  3.5  18   23   a 2 x 4 matrix.

1   2   4   8       A matrix of one row,
                    a 1 x 4 matrix, also known as a
                    'row vector' of 4 elements.
                    (column vectors are analogously defined)
```

Since matrices consist of a specific number of rows and columns (the *dimensions* of the matrix), which normally do not change when using matrices, we might consider specifying their values as template non-type parameters. Since the `DataType = double` selection will be used in the majority of cases,

double can be selected as the template's default type. Since it's having a sensible default, the DataType template type parameter is put last in the template type parameter list. So, our template class Matrix starts off as follows:

```
template <size_t Rows, size_t Columns, typename DataType = double>
class Matrix
...
```

Various operations are defined on matrices. They may, for example be added, subtracted or multiplied. We will not focus on these operations here. Rather, we'll concentrate on a simple operation: computing marginals and sums. The row marginals are obtained by computing, for each row, the sum of all its elements, putting these Rows sum values in corresponding elements of a column vector of Rows elements. Analogously, column marginals are obtained by computing, for each column, the sum of all its elements, putting these Columns sum values in corresponding elements of a row vector of Columns elements. Finally, the sum of the elements of a matrix can be computed. This sum is of course equal to the sum of the elements of its marginals. The following example shows a matrix, its marginals, and its sum:

```
            matrix:         row
                            marginals:

            1   2   3        6
            4   5   6       15

column      5   7   9       21  (sum)
marginals
```

So, what do we want our class template to offer?

- It needs a place to store its matrix elements. This can be defined as an array of 'Rows' rows each containing 'Columns' elements of type DataType. It can be an array, rather than a pointer, since the matrix' dimensions are known *a priori*. Since a vector of Columns elements (a *row* of the matrix), as well as a vector of Row elements (a *column* of the matrix) is often used, *typedefs* could be used by the class. The class interface's initial section therefore contains:

```
typedef Matrix<1, Columns, DataType>    MatrixRow;
typedef Matrix<Rows, 1, DataType>       MatrixColumn;

MatrixRow d_matrix[Rows];
```

- It should offer constructors: a default constructor and, for example, a constructor initializing the matrix from a stream. No copy constructor is required, since the default copy constructor performs its task properly. Analogously, no overloaded assignment operator or destructor is required. Here are the constructors, defined in the public section:

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, Columns, DataType>::Matrix()
{
    std::fill(d_matrix, d_matrix + Rows, MatrixRow());
}

template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, Columns, DataType>::Matrix(std::istream &str)
{
    for (size_t row = 0; row < Rows; row++)
        for (size_t col = 0; col < Columns; col++)
            str >> d_matrix[row][col];
}
```

- The class's `operator[]()` member (and its `const` variant) only handles the first index, returning a reference to a complete `MatrixRow`. How to handle the retrieval of elements in a `MatrixRow` will be covered shortly. To keep the example simple, no array bound check has been implemented:

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, Columns, DataType>
&Matrix<Rows, Columns, DataType>::operator[](size_t idx)
{
    return d_matrix[idx];
}
```

- Now we get to the interesting parts: computing marginals and the sum of all elements in a `Matrix`. Considering that marginals are vectors, either a `MatrixRow`, containing the column marginals, a `MatrixColumn`, containing the row marginals, or a single value, either computed as the sum of a vector of marginals, or as the value of a `1 x 1` matrix, initialized from a generic `Matrix`, we can now construct *partial specializations* to handle `MatrixRow` and `MatrixColumn` objects, and a partial specialization handling `1 x 1` matrices. Since we're about to define these specializations, we can use them when computing marginals and the matrix' sum of all elements. Here are the implementations of these members:

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, Columns, DataType>
Matrix<Rows, Columns, DataType>::columnMarginals() const
{
    return MatrixRow(*this);
}

template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, 1, DataType>
Matrix<Rows, Columns, DataType>::rowMarginals() const
{
    return MatrixColumn(*this);
}

template <size_t Rows, size_t Columns, typename DataType>
DataType Matrix<Rows, Columns, DataType>::sum()  const
{
    return rowMarginals().sum();
}
```

Class template *partial specializations* may be defined for any (subset) of template parameters. They can be defined for template type parameters and for template non-type parameters alike. Our first partial specialization defines the special case where we construct a row of a generic `Matrix`, specifically aiming at (but not restricted to) the construction of column marginals. Here is how such a partial specialization is constructed:

- The partial specialization starts by defining all template type parameters which are *not* specialized in the partial specialization. This partial specialization template announcement cannot specify any defaults (like `DataType = double`), since the defaults have already been specified by the generic class template definition. Furthermore, the specialization *must* follow the definition of the generic class template definition, or the compiler will complain that it doesn't know what class is being specialized. Following the template announcement, the class interface starts. Since it's a class template (partial) specialization, the class name is followed by a template type parameter list specifying ordinary values or types for all template parameters specified in this specialization, and using the template's generic (non-)type names for the remaining template parameters. In the `MatrixRow` specialization `Rows` is specified as 1, since we're talking here about one single row. Both `Columns` and `DataType` remain to be specified. So, the `MatrixRow` partial specialization starts as follows:

```
template <size_t Columns, typename DataType>  // no default specified
```

```
class Matrix<1, Columns, DataType>
```

- A `MatrixRow` contains the data of a single row. So it needs a data member storing `Columns` values of type `DataType`. Since `Columns` is a constant value, the `d_row` data member can be defined as an array:

```
DataType d_column[Columns];
```

- The constructors require some attention. The default constructor is simple. It merely initializes the `MatrixRow`'s data elements using `DataType`'s default constructor:

```
template <size_t Columns, typename DataType>
Matrix<1, Columns, DataType>::Matrix()
{
    std::fill(d_column, d_column + Columns, DataType());
}
```

  However, we also need a constructor initializing a `MatrixRow` object with the column marginals of a generic `Matrix` object. This requires us to provide the constructor with a non-specialized `Matrix` parameter. In cases like this, the rule of thumb is to define a member template allowing us to keep the general nature of the parameter. Since the generic `Matrix` template requires three template parameters, two of which are already provided by the template specialization, the third parameter must be specified in the member template's template announcement. Since this parameter refers to the generic matrix' number of rows, let's simply call it `Rows`. Here then, is the definition of the second constructor, initializing the `MatrixRow`'s data with the column marginals of a generic `Matrix` object:

```
template <size_t Columns, typename DataType>
template <size_t Rows>
Matrix<1, Columns, DataType>::Matrix(
                        Matrix<Rows, Columns, DataType> const &matrix)
{
    std::fill(d_column, d_column + Columns, DataType());

    for (size_t col = 0; col < Columns; col++)
        for (size_t row = 0; row < Rows; row++)
            d_column[col] += matrix[row][col];
}
```

  Note the way the constructor's parameter is defined: it's a reference to a `Matrix` template using the additional `Row` template parameter as well as the template parameters of the partial specialization itself.

- We don't really require additional members to satisfy our current needs. To access the data elements of the `MatrixRow` an overloaded `operator[]()` is of course useful. Again, the `const` variant can be implemented like the non-`const` variant. Here is its implementation:

```
template <size_t Columns, typename DataType>
DataType &Matrix<1, Columns, DataType>::operator[](size_t idx)
{
    return d_column[idx];
}
```

Now that we have defined the generic `Matrix` class as well as the partial specialization defining a single row, the compiler will select the row's specialization whenever a `Matrix` is defined using `Row = 1`. For example:

```
Matrix<4, 6> matrix;        // generic Matrix template is used
Matrix<1, 6> row;           // partial specialization is used
```

The partial specialization for a MatrixColumn is constructed similarly. Let's present its highlights (the full Matrix class template definition as well as all its specializations are provided in the cplusplus.yo.zip archive (at SourceForge[1]) in the file yo/classtemplates/examples/matrix.h):

- The class template partial specialization again starts with a template announcement. The class definition itself now specifies a fixed value for the second (generic) template parameter, illustrating that we can construct partial specializations for every single template parameter; not just the first or the last:

```
template <size_t Rows, typename DataType>
class Matrix<Rows, 1, DataType>
```

- Its constructors are implemented completely analogously to the way the MatrixRow constructors were implemented. Their implementations are left as an exercise to the reader (and they can be found in matrix.h).
- An additional member sum() is defined to compute the sum of the elements of a MatrixColumn vector. It's simply implemented using the accumulate() generic algorithm:

```
template <size_t Rows, typename DataType>
DataType Matrix<Rows, 1, DataType>::sum()
{
    return std::accumulate(d_row, d_row + Rows, DataType());
}
```

The reader might wonder what happens if we specify the following matrix:

```
Matrix<1, 1> cell;
```

Is this a MatrixRow or a MatrixColumn specialization? The answer is: neither. It's ambiguous, precisely because *both* the columns *and* the rows could be used with a (different) template partial specialization. If such a Matrix is actually required, yet another specialized template must be designed. Since this template specialization can be useful to obtain the sum of the elements of a Matrix, it's covered here as well:

- This class template partial specialization also needs a template announcement, this time only specifying DataType. The class definition specifies two fixed values using 1 for both the number of rows and the number of columns:

```
template <typename DataType>
class Matrix<1, 1, DataType>
```

- The specialization defines the usual batch of constructors. Again, constructors expecting a more generic Matrix type are implemented as member templates. For example:

```
template <typename DataType>
template <size_t Rows, size_t Columns>
Matrix<1, 1, DataType>::Matrix(
                         Matrix<Rows, Columns, DataType> const &matrix)
:
    d_cell(matrix.rowMarginals().sum())
{}

template <typename DataType>
template <size_t Rows>
Matrix<1, 1, DataType>::Matrix(Matrix<Rows, 1, DataType> const &matrix)
:
    d_cell(matrix.sum())
{}
```

[1] http://sourceforge.net/projects/cppannotations/

- Since `Matrix<1, 1>` is basically a wrapper around a `DataType` value, we need members to access that latter value. A type conversion operator might be usefull, but we'll also need a `get()` member to obtain the value if the conversion operator isn't used by the compiler (which happens when the compiler is given a choice, see section 10.3). Here are the accessors (leaving out their `const` variants):

```
template <typename DataType>
Matrix<1, 1, DataType>::operator DataType &()
{
    return d_cell;
}

template <typename DataType>
DataType &Matrix<1, 1, DataType>::get()
{
    return d_cell;
}
```

The following `main()` function shows how the `Matrix` class template and its partial specializations can be used:

```
#include <iostream>
#include "matrix.h"
using namespace std;

int main(int argc, char **argv)
{
    Matrix<3, 2> matrix(cin);

    Matrix<1, 2> colMargins(matrix);
    cout << "Column marginals:\n";
    cout << colMargins[0] << " " << colMargins[1] << endl;

    Matrix<3, 1> rowMargins(matrix);
    cout << "Row marginals:\n";
    for (size_t idx = 0; idx < 3; idx++)
        cout << rowMargins[idx] << endl;

    cout << "Sum total: " << Matrix<1, 1>(matrix) << endl;
    return 0;
}
/*
    Generated output from input: 1 2 3 4 5 6

    Column marginals:
    9 12
    Row marginals:
    3
    7
    11
    Sum total: 21
*/
```

21.5 Variadic templates (C++0x)

Up to this point we've only encountered templates having a fixed number of template parameters. The C++0x standard extends this with *variadic templates*.

Variadic templates are defined for function templates and for class templates. Variadic templates allow the specification of an arbitrary number of arguments of any type.

They were added to the language to prevent the template constructor from having to define many overloaded template versions and to be able to create *type safe* variadic functions.

Although **C** (and **C++**) support variadic functions, their use has always been deprecated in **C++** as those functions are notoriously type-*un*safe. Furthermore, variadic function templates can be used to allow these functions to process objects that until now couldn't be processed properly by traditional variadic functions.

Here is an example of the declaration of a variadic template `Variadic`:

```
template<typename... Values> class Variadic;
```

Assuming the template's definition has been provided then this template can be instantiated using any number of typenames. E.g.,

```
class Variadic<
        int,
        std::vector<int>,
        std::map<std::string, std::vector<int>>
> v1;
```

It is possible to define an 'empty' variadic template by not specifying any type when instantiating the variadic template. E.g.,

```
class Variadic<> empty;
```

On the other hand, the instantiation of an empty variadic template may be prevented by providing one or more fixed parameters. E.g.,

```
template<typename First, typename... Rest>
class tuple;
```

Here is an example of the declaration of a variadic function template `printf`:

```
template<typename ... Params>
void printf(std::string const &strFormat, Params ... parameters);
```

Note the use of the ellipsis (...). It serves two purposes:

- Written to the *left* of a template parameter name, it declares a *parameter pack*. A parameter pack allows the software engineer to associate arguments (maybe none) with the variadic template parameters. Parameter packs can be used to bind type and non-type template parameters.
- Written to the *right* of a template or function call argument, it represents a series of template arguments.

Although no syntax is offered to determine the individual parameters they can be determined recursively. An example is provided in the next section. In addition the number of arguments in a template parameter pack can be determined using `sizeof` as shown in the following example:

```
template<typename ... Args>
struct StructName
{
    static size_t const s_size = sizeof...(Args);
};

// SomeName<int, char>::s_size          -- returns 2
```

21.5.1 Defining and using variadic templates (C++0x)

As the variadic parameters themselves are not directly available to the implementation of a function or class the software engineer must resort to other means to obtain them. Typically a recursive definition is used decaying to a non-argument implementation once all parameters have been processed.

E.g., a variadic template definition of `printf` might process all template parameters and then call the following `printf` implementation to print the remainder of the format string, *en passant* checking for any left-over format specifications:

```
void printf(char const *s)
{
    while (*s)
    {
        if (*s == '%' && *(++s) != '%')
            throw std::runtime_error(
                        "invalid format string: missing arguments");
        std::cout << *s++;
    }
}
```

Please don't let the `printf` function name confuse you: it's not **C**. Note that the type's value is inserted into the output stream by `cout` (line 8): type safety and extensibility is guaranteed.

The implementation of `printf` itself is recursive. The function's template parameter list (line 1) specifies an initial type followed by a variadic type list, and only that type's value is directly processed by the function's body (in line 8).

At line 9 the function calls itself, passing the remaining values to its recursive call. A subtlety in the recursive call allows for the detection of any remaining types, not accounted for by the format specification list. In that case the function will throw an exception. Also note that the format string differs somewhat from what is used by **C** (line 6). The character following the % format specifier is ignored unless it is another %, in which case a % is printed. Since `cout` takes care of the type safety there's no need to specify exactly what the type of a particular argument is. Extending the format specifiers so that field widths etc. are recognized by this `printf` implementation is left as an exercise to the Annotation's reader.

```
template<typename T, typename ... Args>                 // 1
void printf(const char* s, T value, Args ... args)      // 2
{
    while (*s)
    {
        if (*s == '%' && *(++s) != '%')                 // 6
        {
            std::cout << value;                         // 8
            printf(*s ? ++s : s, args...);              // 9
            return;
        }
        std::cout << *s++;
    }
    throw std::logic_error("extra arguments provided to printf");
}
```

21.5.2 Perfect forwarding (C++0x)

Rvalue references, introduced in **C++** by the C++0x standard can also be used to achieve *perfect forwarding*, which was not available in **C++** before the advent of the C++0x standard. In combination with

variadic templates rvalue references allow function templates to forward arguments to other functions accepting such arguments. The provided arguments are 'perfectly forwarded' as they are forwarded in a type-safe way: called functions must have matching arguments, matching both in types and number, in order to be called.

Perfect forwarding is very useful in situations where objects must pass on arguments to (overloaded) members of, e.g., composed objects. Traditionally, a class intending to expose some of the functionality of a composed member had to repeat that part of the composed member's interface in its own interface. Perfect forwarding may change this and may only require the addition of one single member template to the composing class's interface. In the following example a class `Composer` wants to expose the `substr` members of its composed `std::string d_str`. Rather than to repeat all overloaded versions of `std::string` in its own interface it uses perfect forwarding to be able to call all overloaded `substr` members of `d_str`:

```
class Composer
{
    string d_str;   // somehow initialized, e.g., via a constructor

    public:
        template<typename ...Args>
        string substr(Args&&... args)
        {
            return d_cl.substr(std::forward<Args>(args)...);
        }
};
```

The core elements in the above example are:

- The definition of `substr` as a variadic template
- *Forwarding* the function's arguments, keeping track of their types, using `std::forward<Args>(args) ...`.

As a result, the version of `d_str.substr` that is actually called depends on the arguments passed to `Composer::substr`.

Another situation where perfect forwarding is useful is where a factory function is to be constructed. A factory function forwards its arguments to constructors of other classes, returning the thus constructed object. When a factory function also receives the typename of the class to construct it can be constructed as a generic factory function. A factory function may be useful when the construction of several objects of several types must be centralized (e.g., when object construction should be logged or objects should be constructed dynamically and shared pointers are used to refer to constructed objects). Here is an example of a generic factory function that may be used to obtain any kind of object using whatever set of arguments the constructors may expect:

```
template <typename Class, typename ... Args>
Class factory(Args&& ... args)
{
    return Class(std::forward<Args>(args) ...);
}
```

This example shows the same core elements as encountered previously. A variadic function template forwarding its argument to a constructor of a class specified at call-time. Assume the existence of two classes `One` and `Two`, each having a default constructor, a constructor expecting an `int` and a copy constructor, then objects could be constructed by the factory as follows:

```
One one(factory<One>());
Two two(factory<Two>(3));
Two second(factory<Two>(two));
```

Here a default `One` object, a `Two` object using `Two`'s constructor expecting and `int` and a `Two` object using `Two`'s copy constructor are constructed.

21.5.3 The unpack operator (C++0x)

The *unpack operator* can be used to obtain template arguments in many situations. No mechanism other than recursion (as shown in the previous section) is available to obtain the individual types or values of a variadic template.

The unpack operator can also be used to define a template class that is derived from any number of base classes. Here is an example:

```
template <typename... BaseClasses>
class ClassName: public BaseClasses ...
{
    public:
        ClassName(BaseClasses && ... baseClasses)
        :
            BaseClasses(baseClasses) ...
        {}
};

    // Define a class derived from a vector, a complex and a string:
ClassName<std::vector<std::string>>, std::complex, std::string>
    object({"one", "two", "three"}, {3, 2.5}, "hello world");
```

In this example the unpack operator appears as the ellipses trailing `BaseClasses`. The class `ClassName` is derived from each of the types specified when instantiating `ClassName` and here `ClassName`'s constructor expects a reference to each base class which is used to initialize the base classes of ClassName. The construction of `object` illustrates how this may be accomplished: (uniform) intializers are used in a sequence matching the sequence of the types specified for the used `ClassName` type.

When using the unpack operator in combination with function templates it allows the template to forward the variadic parameters. When rvalue references are used in this case perfect forwarding (cf. section 21.5.2) can be used. Here is an example:

```
template<typename Type>
struct Allocator
{
    template<typename ... Args>
    std::shared_ptr<Type> factory(Args && ... params)
    {
        return std::shared_ptr<Type>(
                        new Type(std::forward<Args>(params) ...));
    }
}
```

This example has the following characteristics:

- The argument list of the member function template `factory` is unpacked into `Type`'s constructor.
- The `std::forward<Args>(params)` syntax ensures that an argument is forwared to `Type`'s constructor using its actual type (and a matching `Type` constructor must be available).
- In addition to this (i.e., to using `forward`) the unpack operator (activated by the ellipses trailing the `std::forward` call) ensures that the forwarding is applied to each argument in turn.

21.5.4 Tuples (C++0x)

The C++0x standard offers a *generalized pair* container: the *tuple*, which is covered in this section. Whereas `std::pair` containers have limited functionality and only two members, tuples have slightly more functionality and may consist of a unlimited number of different data types. In that respect a tuple can be thought of a the 'template's answer to **C**'s struct'.

A tuple's generic definition uses a variadic template notation:

```
template <class ...Types> class tuple;
```

Here is an example of its actual definition and use:

```
typedef std::tuple<int, double &, std::string, char const *> tuple_idsc;

double pi = 3.14;
tuple_idsc idsc(59, pi, "hello", "fixed");

// access a field:
std::get<2>(idsc) = "hello world";
```

The template argument to the `std::get<idx>` function template is an index on the ordered set of template types defined for the tuple, with 0 returning the value of its first type.

Tuples may be constructed without specifying initial values. For primitive types 0 will be used, for class types their default constructors. However, note that the construction may succeed but its use may fail. Consider:

```
tuple<int &> empty;
cout << get<0>(empty);
```

The tuple `empty` cannot be used as its `int &` field is an undefined reference. `Empty`'s construction will succeed, though.

Tuples may be assigned to each other if their types are identical; if appropriate copy constructors are defined; if a right-hand type can be converted to its matching left-hand type; or if the left-hand type can be constructed from the matching right-hand type. Furthermore, tuples (matching in number and (convertible) types) can be compared using relational operators.

Finally the following static elements are defined for tuple types (using compile-time initialization):

- `std::tuple_size<Tuple>::value` returns the number of types defined for the tuple type `Tuple`, e.g. the following statement displays 4:

  ```
  cout << tuple_size<tuple_idsc>::value << endl;
  ```

- `std::tuple_element<idx, Typle>::type` returns the type of element `idx` of `Tuple`. E.g., the following defines a `string` variable:

  ```
  tuple_element<2, tuple_idsc>::type text;
  ```

To use the `tuple` template the header file `tuple` must be included.

21.5.5 User-defined literals (C++0x)

The C++0x standard offers *extensible literals*, also known as *user defined literals*. Standard **C++** defines various kinds of literals, like numerical constants (with or without suffixes), character constants and string (textual) literals.

Under the C++0x standard a literal, e.g., 1013, can be looked at in two ways: as a *raw literal* in which the literal as ASCII string or series of characters is passed to a processing function or as a *cooked literal* in which case the compiler already performs some conversion.

To extend a literal either a processing function or a variadic function template must be defined. Both functions must start with an underscore and all user defined literals are suffixes.

To declare a (raw) literal the following form is used:

```
Type operator "" _functionName(char const *text [, size_t len]);
```

Here, `Type` is the resulting type of the literal, and '`, size_t len`' is an optional parameter receiving `text`'s length (as returned by **C**'s `strlen` function)

The function `_functionName` must be implemented by the software engineer and must return a value of type `Type`. Here is an example of an user defined literal `_kmh` returning a `double` and how it can be used:

```
double _kmh(char const *speed)
{
    std::istringstream in(speed);
    double ret;
    in >> ret;
    return ret;
}
double operator "" _kmh(char const *speed);

double value = "120"_kmh;
```

No information is available as to when user defined literals will be supported by the `g++` compiler. Once this information becomes available this section will be updated with examples showing how extensible literals can be used.

21.6 Template typedefs and 'using' syntax (C++0x)

Standard C++ allows the use of `typedefs` to create 'shorthand' notations for extensive type names, often resulting in code that is easier to write and usually easier to read and understand. E.g., instead of writing

```
std::map<std::shared_ptr<KeyType>, std::vector<std::set<std::string>>>
    mapObject;
```

it is possible to write

```
MapKeyVectorStringSet mapObject;
```

after having defined

```
typedef
   std::map<std::shared_ptr<KeyType>, std::vector<std::set<std::string>>>
        MapKeyVectorStringSet;
```

However, it may be that the data type varies. Multiple maps might be used, some of them storing `string` objects, other might store `double` values. A class may be defined inheriting from this complex map allowing the specification of, e.g., the final key and value types. E.g.,

```
template <typename DeepKey, typename DeepValue>
```

```
class DeepMap: public std::map<std::shared_ptr<DeepKey>,
                        std::vector<std::set<DeepMap>>>
{};
```

This doubtlessly reduces the amount of work, as it is now possible to write

```
DeepMap<KeyType, std::string> mapObject;
```

The C++0x standard adds to this the possibility to become even lazier: it allows the definition of a `typedef` in which some of the template parameters are specified leaving other parameters to be specified at a later moment in time. E.g., the C++0x standard allows fixating the key type and leaving the value type open by introducing the following syntax:

```
template< typename DeepValue>
using MapKeyDeepValue = DeepMap<KeyType, DeepValue>;

MapKeyDeepValue<std::string> sameTypeAs(mapObject);
```

a similar shorthand notation could be used to fixate the value type and leave the key type open for later specification.

In addition to the above *templated typedef's* the `using` keyword can also be used to separate a typedef name from a complex type definition. Consider constructing a typedef for a pointer to a function expecting and returning a `double`. The traditional way to define such a type is as follows:

```
typedef double (*PFD)(double);
```

The C++0x standard defines the following alternative syntax:

```
using PFD = double (*)(double);
```

Partially specified typedefs and the alternative syntax for type definition are not yet supported by the `g++` compiler.

21.7 Instantiating class templates

Template classes are instantiated when an object of a class template is defined. When a class template object is defined or declared, the template parameters must explicitly be specified.

Template parameters are *also* specified when a class template defines default template parameter values, albeit that in that case the compiler will provide the defaults (cf. section 21.4 where `double` is used as the default type to be used with the template's `DataType` parameter). The actual values or types of template parameters are *never* deduced, as happens with function templates: to define a `Matrix` of elements that are complex values, the following construction is used:

```
Matrix<3, 5, std::complex> complexMatrix;
```

while the following construction defines a matrix of elements that are `double` values, with the compiler providing the (default) type `double`:

```
Matrix<3, 5> doubleMatrix;
```

A class template object may be *declared* using the keyword `extern`. For example, the following construction is used to *declare* the matrix `complexMatrix`:

```
extern Matrix<3, 5, std::complex> complexMatrix;
```

A class template declaration is sufficient if the compiler encounters function declarations of functions having return values or parameters which are class template objects, pointers or references. The following little source file may be compiled, although the compiler hasn't seen the definition of the `Matrix` class template. Note that generic classes as well as (partial) specializations may be declared. Furthermore, note that a function expecting or returning a class template object, reference, or parameter itself automatically becomes a function template. This is necessary to allow the compiler to tailor the function to the types of various actual arguments that may be passed to the function:

```
#include <stddef.h>

template <size_t Rows, size_t Columns, typename DataType = double>
class Matrix;

template <size_t Columns, typename DataType>
class Matrix<1, Columns, DataType>;

Matrix<1, 12> *function(Matrix<2, 18, size_t> &mat);
```

When class templates are used they have to be processed by the compiler first. So, template member functions must be known to the compiler when the template is instantiated. This does not mean that all members of a template class are instantiated when a class template object is defined. The compiler will only instantiate those members that are actually used. This is illustrated by the following simple class `Demo`, having two constructors and two members. When we create a `main()` function in which one constructor is used and one member is called, we can make a note of the sizes of the resulting object file and executable program. Next the class definition is modified such that the unused constructor and member are commented out. Again we compile and link the `main()` function and the resulting sizes are identical to the sizes obtained earlier (on my computer, using `g++` version 4.1.2) these sizes are 3904 bytes (after stripping). There are other ways to illustrate the point that only members that are used are instantiated, like using the `nm` program, showing the symbolic contents of object files. Using programs like `nm` will yield the same conclusion: *only template member functions that are actually used are initialized*. Here is an example of the class template `Demo` used for this little experiment. In `main()` only the first constructor and the first member function are called and thus only these members were instantiated:

```
#include <iostream>

template <typename Type>
class Demo
{
    Type d_data;

    public:
        Demo();
        Demo(Type const &value);

        void member1();
        void member2(Type const &value);
};

template <typename Type>
Demo<Type>::Demo()
:
    d_data(Type())
{}

template <typename Type>
void Demo<Type>::member1()
{
```

```
    d_data += d_data;
}

// the following members are commented out before compiling
// the second program

template <typename Type>
Demo<Type>::Demo(Type const &value)
:
    d_data(value)
{}

template <typename Type>
void Demo<Type>::member2(Type const &value)
{
    d_data += value;
}

int main()
{
    Demo<int> demo;
    demo.member1();
}
```

21.8 Processing class templates and instantiations

In section 20.11 the distinction between code depending on template parameters and code not depending on template parameters was introduced. The same distinction also holds true when class templates are defined and used.

Code that does not depend on template parameters is verified by the compiler when the template is defined. E.g., if a member function in a class template uses a `qsort()` function, then `qsort()` does not depend on a template parameter. Consequently, `qsort()` must be known to the compiler when it encounters the `qsort()` function call. In practice this implies that `cstdlib` or `stdlib.h` must have been processed by the compiler before it will be able to process the class template definition.

On the other hand, if a template defines a `<typename Type>` template type parameter, which is the return type of some template member function, e.g.,

```
Type member() ...
```

then we distinguish the following situations where the compiler encounters `member()` or the class to which `member()` belongs:

- At the location in the source where class template objects are defined (called the *point of instantiation* of the class template object), the compiler will have read the class template definition, performing a basic check for syntactic correctness of member functions like `member()`. So, it won't accept a definition or declaration like `Type &&member()`, because **C++** does not support functions returning references to references. Furthermore, it will check the existence of the actual typename that is used for instantiating the object. This typename must be known to the compiler at the object's point of instantiation.
- At the location in the source where template member functions are used (which is called the template member function's point of instantiation), the `Type` parameter must of course still be known, and `member()`'s statements that depend on the `Type` template parameter are now checked for syntactic correctness. For example, if `member()` contains a statement like

```
Type tmp(Type(), 15);
```

then this is in principle a syntactically valid statement. However, when `Type = int` and `member()` is called, its instantiation will fail, because `int` does not have a constructor expecting two `int` arguments. Note that this is *not* a problem when the compiler instantiates an object of the class containing `member()`: at the point of instantiation of the object its `member()` member function is not instantiated, and so the invalid `int` construction remains undetected.

21.9 Declaring friends

Friend functions are normally constructed as *support* functions of a class that cannot be constructed as class members themselves. The well-known insertion operator for output streams is a case in point. Friend classes are most often seen in the context of nested classes, where the inner class declares the outer class as its friend (or the other way around). Here again we see a support mechanism: the inner class is constructed to support the outer class.

Like ordinary classes, class templates may declare other functions and classes as their friends. Conversely, ordinary classes may declare template classes as their friends. Here too, the friend is constructed as a special function or class augmenting or supporting the functionality of the declaring class. Although the `friend` keyword can thus be used in any type of class (ordinary or template) to declare any type of function or class as a friend, when using class templates the following cases should be distinguished:

- A class template may declare an ordinary function or class to be its friend. This is a common friend declaration, such as the insertion operator for `ostream` objects.
- A class template may declare another function template or class template to be its friend. In this case, the friend's template parameters may have to be specified. If the actual values of the friend's template parameters must be equal to the template parameters of the class declaring the friend, the friend is said to be a *bound friend template* class or function. In this case the template parameters of the template in which a `friend` declaration is used determine (*bind*) the template parameters of the friend class or function, resulting in a one-to-one correspondence between the template's parameters and the friend's template parameters.
- In the most general case, a class template may declare another function template or class template to be its friend, irrespective of the friend's actual template parameters. In this case an *unbound friend template* class or function is declared: the template parameters of the friend class template or function remain to be specified, and are not related in some predefined way to the template parameters of the class declaring the friend. For example, if a class has data members of various types, specified by its template parameters, and another class should be allowed direct access to these data members (so it should be a friend), we would like to specify any of the current template parameters to instantiate such a friend. Rather than specifying multiple bound friends, a single generic (unbound) friend may be declared, specifying the friend's actual template parameters only when this is required.
- The above cases, in which a template is declared as a friend, may also be encountered when ordinary classes are used:
  - The ordinary class declaring ordinary friends has already been covered (chapter 15).
  - The equivalent of bound friends occurs if a ordinary class specifies specific actual template parameters when declaring its friend.
  - The equivalent of unbound friends occurs if a ordinary class declares a generic template as its friend.

21.9.1 Non-function templates or classes as friends

A class template may declare a ordinary function, ordinary member function or complete ordinary class as its friend. Such a friend may access the class template's private members.

Concrete classes and ordinary functions can be declared as friends, but before a single class member function can be declared as a friend, the compiler must have seen the class interface declaring that member. Let's consider the various possibilities:

- A class template may declare a ordinary function to be its friend. It is not completely clear *why* we would like to declare a ordinary function as a friend. In ordinary cases we would like to pass an object of the class declaring the friend to the function. However, this requires us to provide the function with a template parameter without specifying its types. As the language does not support constructions like

  ```
  void function(std::vector<Type> &vector)
  ```

  unless `function()` itself is a template, it is not immediately clear how and why such a friend should be constructed. One reason, though, is to allow the function to access the class's private static members. Furthermore, such friends could themselves instantiate objects of classes declaring them as friends, and directly access such object's private members. For example:

  ```
  template <typename Type>
  class Storage
  {
      friend void basic();
      static size_t s_time;
      std::vector<Type> d_data;
      public:
          Storage();
  };

  template <typename Type>
  size_t Storage<Type>::s_time = 0;

  template <typename Type>
  Storage<Type>::Storage()
  {}

  void basic()
  {
      Storage<int>::s_time = time(0);
      Storage<double> storage;
      std::random_shuffle(storage.d_data.begin(), storage.d_data.end());
  }
  ```

- Declaring a ordinary class to be a class template's friend probably has more practical implications. Here the friend-class may instantiate any kind of object of the class template, to access all of its private members thereafter. A simple forward declaration of the friend class in front of the class template definition is enough to make this work:

  ```
  class Friend;

  template <typename Type>
  class Composer
  {
      friend class Friend;
      std::vector<Type> d_data;
      public:
          Composer();
  };

  class Friend
  {
  ```

```
    Composer<int> d_ints;
    public:
        Friend(std::istream &input);
};

inline::Friend::Friend(std::istream &input)
{
    std::copy(std::istream_iterator<int>(input),
              std::istream_iterator<int>(),
              back_inserter(d_ints.d_data));
}
```

- Alternatively, just a single member function of a ordinary class may be declared as a friend. This requires that the compiler has read the friend class's interface before the friend is declared. Omitting the required destructor and overloaded assignment operators, the following shows an example of a class whose member `randomizer()` is declared as a friend of the class `Composer`:

```
template <typename Type>
class Composer;

class Friend
{
    Composer<int> *d_ints;
    public:
        Friend(std::istream &input);
        void randomizer();
};

template <typename Type>
class Composer
{
    friend void Friend::randomizer();
    std::vector<Type> d_data;
    public:
        Composer(std::istream &input)
        {
            std::copy(std::istream_iterator<int>(input),
                      std::istream_iterator<int>(),
                      back_inserter(d_data));
        }
};

inline Friend::Friend(std::istream &input)
:
    d_ints(new Composer<int>(input))
{}

inline void Friend::randomizer()
{
    std::random_shuffle(d_ints->d_data.begin(), d_ints->d_data.end());
}
```

In this example note that `Friend::d_ints` is a pointer member. It cannot be a `Composer<int>` object, since the `Composer` class interface hasn't yet been seen by the compiler when it reads `Friend`'s class interface. Disregarding this and defining a data member `Composer<int> d_ints` results in the compiler generating the error

```
    error: field 'd_ints' has incomplete type
```

Incomplete type, as the compiler at this points knows of the existence of the class `Composer` but as it hasn't seen `Composer`'s interface it doesn't know what size the `d_ints` data member will have.

21.9.2 Templates instantiated for specific types as friends

With *bound friend* class templates or functions there is a one-to-one mapping between the actual values of the template-friends' template parameters and the class template's template parameters declaring them as friends. In this case, the friends themselves are templates too. Here are the various possibilities:

- A function template may be declared as a friend of a template class. In this case we don't experience the problems we encountered with ordinary functions declared as friends of class templates. Since the friend function template itself is a template, it may be provided with the required template parameters allowing it to specify a class template parameter. Thus we can pass an object of the class declaring the template function as its friend to the function template. The organization of the various declarations thus becomes:
  - The class template declaring the friend is itself declared;
  - The function template (to be declared as a friend) is declared;
  - The class template declaring the bound template friend function is defined;
  - The (friend) function template is defined, now having access to all the class template's (private) members.

  Note that the template friend declaration specifies its template parameters immediately following the template's function name. Without the template parameter list affixed to the function name, it would be an ordinary friend function. Here is an example showing the use of a bound friend to create a subset of the entries of a dictionary. For real life examples, a dedicated function object returning `!key1.find(key2)` is probably more useful, but for the current example, `operator==()` is acceptable:

```
template <typename Key, typename Value>
class Dictionary;

template <typename Key, typename Value>
Dictionary<Key, Value>
           subset(Key const &key, Dictionary<Key, Value> const &dict);

template <typename Key, typename Value>
class Dictionary
{
    friend Dictionary<Key, Value> subset<Key, Value>
               (Key const &key, Dictionary<Key, Value> const &dict);
    std::map<Key, Value> d_dict;
    public:
        Dictionary();
};

template <typename Key, typename Value>
Dictionary<Key, Value>
           subset(Key const &key, Dictionary<Key, Value> const &dict)
{
    Dictionary<Key, Value> ret;

    std::remove_copy_if(dict.d_dict.begin(), dict.d_dict.end(),
                        std::inserter(ret.d_dict, ret.d_dict.begin()),
                        std::bind2nd(std::equal_to<Key>(), key));
```

```
    return ret;
}
```

- By declaring a full class template as a class template's friend, all members of the friend class may access all private members of the class declaring the friend. As the friend class only needs to be declared, the organization of the declaration is much easier than when function templates are declared as friends. In the following example a class `Iterator` is declared as a friend of a class `Dictionary`. Thus, the `Iterator` is able to access `Dictionary`'s private data. There are some interesting points to note here:
  - To declare a class template as a friend, that class is simply declared as a class template before it is declared as a friend:

```
template <typename Key, typename Value>
class Iterator;

template <typename Key, typename Value>
class Dictionary
{
    friend class Iterator<Key, Value>;
```

  - However, the friend class'ss interface may already be used, even before the compiler has seen the friend's interface:

```
template <typename Key, typename Value>
template <typename Key2, typename Value2>
Iterator<Key2, Value2> Dictionary<Key, Value>::begin()
{
    return Iterator<Key, Value>(*this);
}
template <typename Key, typename Value>
template <typename Key2, typename Value2>
Iterator<Key2, Value2> Dictionary<Key, Value>::subset(Key const &key)
{
    return Iterator<Key, Value>(*this).subset(key);
}
```

  - Of course, the friend's interface must still be seen by the compiler. Since it's a support class for `Dictionary`, it can safely define a `std::map` data member, which is initialized by its constructor, accessing the `Dictionary`'s private data member `d_dict`:

```
template <typename Key, typename Value>
class Iterator
{
    std::map<Key, Value> &d_dict;

    public:
        Iterator(Dictionary<Key, Value> &dict)
        :
            d_dict(dict.d_dict)
        {}
```

  - The `Iterator` member `begin()` simply returns a `map` iterator. However, since it is not known to the compiler what the instantiation of the map will look like, a `map<Key, Value>::iterator` is a (deprecated) *implicit typename*. To make it an *explicit* typename, simply prefix `typename` to `begin()`'s return type:

```
template <typename Key, typename Value>
typename std::map<Key, Value>::iterator Iterator<Key, Value>::begin()
{
    return d_dict.begin();
}
```

- In the previous example we might decide that only a `Dictionary` should be able to construct an `Iterator`, as `Iterator` is closely tied to `Dictionary`. This can be implemented by defining `Iterator`'s constructor in its private section, and declaring `Dictionary Iterator`'s friend. Consequently, only `Dictionary` can create its own `Iterator`. By declaring `Iterator`'s constructor as a *bound* friend, we ensure that it can only create `Iterator`s using template parameters identical to its own. Here is how it's implemented:

```
template <typename Key, typename Value>
class Iterator
{
    friend Dictionary<Key, Value>::Dictionary();

    std::map<Key, Value> &d_dict;

    Iterator(Dictionary<Key, Value> &dict);

    public:
```

  In this example, `Dictionary`'s constructor is defined as `Iterator`'s friend. Here the friend is a template member. Other members can be declared as a class's friend as well, in which case their prototypes must be used, including the types of their return values. So, assuming that

```
    std::vector<Value> sortValues()
```

  is a member of `Dictionary`, returning a sorted vector of its values, then the corresponding bound friend declaration would be:

```
friend std::vector<Value> Dictionary<Key, Value>::sortValues();
```

Finally, the following basic example can be used as a prototype for situations where bound friends are useful:

```
template <typename T>           // a function
void fun(T *t)                  // template
{
    t->not_public();
};

template <typename X>           // a class template
class A
{                               // fun() is used as
                                // friend bound to A,
                                // instantiated for X,
                                // whatever X may be
    friend void fun<A<X> >(A<X> *);

    public:
        A();

    private:
        void not_public();
};

template <typename X>
A<X>::A()
{
    fun(this);
}
```

```
template <typename X>
void A<X>::not_public()
{}

int main()
{
    A<int> a;

    fun(&a);                    // fun instantiated for
                                // A<int>.
}
```

21.9.3 Unbound templates as friends

When a friend is declared as an *unbound* friend, it merely declares an existing template to be its friend, no matter how it is instantiated. This may be useful in situations where the friend should be able to instantiate objects of class templates declaring the friend, allowing the friend to access the instantiated object's private members. Again, functions, classes and member functions may be declared as unbound friends.

Here are the syntactic conventions declaring unbound friends:

- Declaring an unbound function template as a friend: any instantiation of the function template may instantiate objects of the template class and may access its private members. Assume the following function template has been defined

  ```
  template <typename Iterator, typename Class, typename Data>
  Class &ForEach(Iterator begin, Iterator end, Class &object,
                      void (Class::*member)(Data &));
  ```

  This function template can be declared as an unbound friend in the following class template `Vector2`:

  ```
  template <typename Type>
  class Vector2: public std::vector<std::vector<Type> >
  {
      template <typename Iterator, typename Class, typename Data>
      friend Class &ForEach(Iterator begin, Iterator end, Class &object,
                  void (Class::*member)(Data &));
      ...
  };
  ```

  If the function template is defined inside some namespace, the namespace must be mentioned as well. E.g., assuming that `ForEach()` is defined in the namespace `FBB` its friend declaration becomes:

  ```
  template <typename Iterator, typename Class, typename Data>
  friend Class &FBB::ForEach(Iterator begin, Iterator end, Class &object,
                  void (Class::*member)(Data &));
  ```

  The following example illustrates the use of an unbound friend. The class `Vector2` stores vectors of elements of template type parameter `Type`. Its `process()` member uses `ForEach()` to have its private `rows()` member called, which in turn uses `ForEach()` to call its private `columns()` member. Consequently, `Vector2` uses two instantiations of `ForEach()`, and therefore an unbound friend is appropriate here. It is assumed that `Type` class objects can be inserted into `ostream` objects (the definition of the `ForEach()` function template can be found in the `cplusplus.yo.zip` archive at `http://sourceforge.net/projects/cppannotations/`). Here is the program:

  ```
  template <typename Type>
  ```

```
class Vector2: public std::vector<std::vector<Type> >
{
    typedef typename Vector2<Type>::iterator iterator;

    template <typename Iterator, typename Class, typename Data>
    friend Class &ForEach(Iterator begin, Iterator end, Class &object,
                void (Class::*member)(Data &));
    public:
        void process();

    private:
        void rows(std::vector<Type> &row);
        void columns(Type &str);
};

template <typename Type>
void Vector2<Type>::process()
{
    ForEach<iterator, Vector2<Type>, std::vector<Type> >
            (this->begin(), this->end(), *this, &Vector2<Type>::rows);
}

template <typename Type>
void Vector2<Type>::rows(std::vector<Type> &row)
{
    ForEach(row.begin(), row.end(), *this,
                                    &Vector2<Type>::columns);
    std::cout << std::endl;
}

template <typename Type>
void Vector2<Type>::columns(Type &str)
{
    std::cout << str << " ";
}

using namespace std;

int main()
{
    Vector2<string> c;
    c.push_back(vector<string>(3, "Hello"));
    c.push_back(vector<string>(2, "World"));

    c.process();
}
/*
    Generated output:

    Hello Hello Hello
    World World
*/
```

- Analogously, a full class template may be declared as a friend. This allows all instantiations of the friend's member functions to instantiate the template declaring the friend class. In this case, the class declaring the friend should offer useful functionality to different instantiations (i.e., using different arguments for its template parameters) of its friend class. The syntactic convention is comparable to the convention used when declaring an unbound friend function template:

```
template <typename Type>
```

```
class PtrVector
{
    template <typename Iterator, typename Class>
    friend class Wrapper;      // unbound friend class
};
```

All members of the class template `Wrapper` may now instantiate `PtrVector`s using any actual type for its `Type` template parameter, at the same time allowing `Wrapper`'s instantiation to access all of `PtrVector`'s private members.

- When only some members of a class template need access to the private members of another class template (e.g., the other class template has private constructors, and only some members of the first class template need to instantiate objects of the second class template), then the latter class template may declare only those members of the former class template requiring access to its private members as its friends. Again, the friend class's interface may be left unspecified. However, the compiler must be informed that the friend member's class is indeed a class. A forward declaration of that class must therefore be given as well. In the following example `PtrVector` declares `Wrapper::begin()` as its friend. Note the forward declaration of the class `Wrapper`:

```
template <typename Iterator>
class Wrapper;

template <typename Type>
class PtrVector
{
    template <typename Iterator> friend
        PtrVector<Type> Wrapper<Iterator>::begin(Iterator const &t1);
    ...
};
```

21.10 Class template derivation

Class templates can be used in class derivation as well. When a class template is used in class derivation, the following situations should be distinguished:

- An existing class template is used as base class when deriving a ordinary class. In this case, the resulting class is still partially a class template, but this is somewhat hidden from view when an object of the derived class is constructed.
- An existing class template is used as the base class when deriving another class template. Here the template-class characteristics remain clearly visible.
- A ordinary class is used as the base class when deriving a template class. This interesting hybrid allows us to construct class templates that are *partially precompiled*.

These three variants of class template derivation will now be elaborated.

Consider the following base class:

```
template<typename T>
class Base
{
    T const &t;

    public:
        Base(T const &t);
};
```

The above class is a class template, which can be used as a base class for the following derived class template `Derived`:

```
template<typename T>
class Derived: public Base<T>
{
    public:
        Derived(T const &t);
};

template<typename T>
Derived<T>::Derived(T const &t)
:
    Base(t)
{}
```

Other combinations are possible as well: by specifying ordinary template type parameters of the base class, the base class is instantiated and the derived class becomes an ordinary class:

```
class Ordinary: public Base<int>
{
    public:
        Ordinary(int x);
};

inline Ordinary::Ordinary(int x)
:
    Base(x)
{}

// With the following object definition:
Ordinary
    o(5);
```

This construction allows us in a specific situation to add functionality to a class template, without the need for constructing a derived class template.

Class template derivation pretty much follows the same rules as ordinary class derivation, not involving class templates. However, some subtleties associated with class template derivation may easily cause confusion. In the following sections class derivation involving class templates will be discussed. Some of the examples shown in these sections may contain unexpected statements and expressions, like the use of `this` when members of a template base class are called from a derived class. The 'chicken and egg' problem I encountered here was solved by first discussing the principles of class template derivation; next the subtleties that are part of class template derivation are covered by section 22.1.

21.10.1 Deriving ordinary classes from class templates

When an existing class template is used as a base class for deriving an ordinary class, the class template parameters are specified when defining the derived class's interface. If in a certain context an existing class template lacks a particular functionality, then it may be useful to derive an ordinary class from a class template. For example, although the class `map` can easily be used in combination with the `find_if()` generic algorithm (section 19.1.16) to locate a particular element, it requires the construction of a class and at least two additional function objects of that class. If this is considered too much overhead in a particular context, extending a class template with some tailor-made functionality might be considered.

A program executing commands entered at the keyboard might accept all unique initial abbreviations

of the commands it defines. E.g., the command `list` might be entered as `l, li, lis` or `list`. By deriving a class `Handler` from

```
map<string, void (Handler::*)(string const &cmd)>
```

and defining a `process(string const &cmd)` to do the actual command processing, the program might simply execute the following `main()` function:

```
int main()
{
    string line;
    Handler cmd;

    while (getline(cin, line))
        cmd.process(line);
}
```

The class `Handler` itself is derived from a complex `map`, in which the map's values are pointers to `Handler`'s member functions, expecting the command line entered by the user. Here are `Handler`'s characteristics:

- The class is derived from a `std::map`, expecting the command associated with each command-processing member as its keys. Since `Handler` uses the map merely to define associations between the commands and the processing member functions, we use private derivation here:

```
class Handler: private std::map<std::string,
                        void (Handler::*)(std::string const &cmd)>
```

- The actual association can be defined using static private data members: `s_cmds` is an array of `Handler::value_type` values, and `s_cmds_end` is a constant pointer pointing beyond the array's last element:

```
    static value_type s_cmds[];
    static value_type *const s_cmds_end;
```

- The constructor simply initializes the map from these two static data members. It could be implemented inline:

```
inline Handler::Handler()
:
    std::map<std::string,
                void (Handler::*)(std::string const &cmd)>
    (s_cmds, s_cmds_end)
{}
```

- The member `process()` iterates along the map's elements. Once the first word on the command line matches the initial characters of the command, the corresponding command is executed. If no such command is found, an error message is issued:

```
void Handler::process(std::string const &line)
{
    istringstream istr(line);
    string cmd;
    istr >> cmd;
    for (iterator it = begin(); it != end(); it++)
    {
        if (it->first.find(cmd) == 0)
        {
```

```
                (this->*it->second)(line);
                return;
            }
        }
        cout << "Unknown command: " << line << endl;
    }
```

21.10.2 Deriving class templates from class templates

Although it's perfectly acceptable to derive an ordinary class from a class template, the resulting class of course has limited generality compared to its template base class. If generality is important, it's probably a better idea to derive a class template from a class template. This allows us the extend an existing class template with some additional functionality, like allowing hierarchic sorting of its elements.

The following class `SortVector` is a class template derived from the existing class template `Vector`. However, it allows us to perform a *hierarchic sort* of its elements using any order of any members its data elements may contain. To accomplish this there is but one requirement: the `SortVector`'s data type must have dedicated member functions comparing its members. For example, if `SortVector`'s data type is an object of class `MultiData`, then `MultiData` should implement member functions having the following prototypes for each of its data members which can be compared:

```
    bool (MultiData::*)(MultiData const &rhv)
```

So, if `MultiData` has two data members, `int d_value` and `std::string d_text`, and both may be required for a hierarchic sort, then `MultiData` should offer members like:

```
bool intCmp(MultiData const &rhv);  // returns d_value < rhv.d_value
bool textCmp(MultiData const &rhv); // returns d_text  < rhv.d_text
```

Furthermore, as a convenience it is also assumed that `operator<<` and `operator>>` have been defined for `MultiData` objects, but that assumption as such is irrelevant to the current discussion.

The class template `SortVector` is derived directly from the template class `std::vector`. Our implementation inherits all members from that base class, as well as two simple constructors:

```
template <typename Type>
class SortVector: public std::vector<Type>
{
    public:
        SortVector()
        {}
        SortVector(Type const *begin, Type const *end)
        :
            std::vector<Type>(begin, end)
        {}
```

However, its member `hierarchicSort()` is the actual reason why the class exists. This class defines the hierarchic sort criteria. It expects an array of pointers to member functions of the class indicated by `sortVector`'s template `Type` parameter as well as a `size_t` indicating the size of the array. The array's first element indicates the class's most significant or first sort criterion, the array's last element indicates the class's least significant or last sort criterion. Since the `stable_sort()` generic algorithm was designed explicitly to support hierarchic sorting, the member uses this generic algorithm to sort `SortVector`'s elements. With hierarchic sorting, the least significant criterion should be sorted first. `hierarchicSort()`'s implementation therefore, is easy, assuming the existence of a support class `SortWith` whose objects are initialized by the addresses of the member functions passed to the `hierarchicSort()` member:

```
template <typename Type>
class SortWith
{
   bool (Type::*d_ptr)(Type const &rhv) const;
```

The class `SortWith` is a simple *wrapper class* around a pointer to a predicate function. Since it's dependent on `SortVector`'s actual data type `SortWith` itself is also a class template:

```
template <typename Type>
class SortWith
{
   bool (Type::*d_ptr)(Type const &rhv) const;
```

It's constructor receives such a pointer and initializes the class's `d_ptr` data member:

```
template <typename Type>
SortWith<Type>::SortWith(bool (Type::*ptr)(Type const &rhv) const)
:
    d_ptr(ptr)
{}
```

Its binary predicate member `operator()()` should return `true` if its first argument should be sorted before its second argument:

```
template <typename Type>
bool SortWith<Type>::operator()(Type const &lhv, Type const &rhv) const
{
    return (lhv.*d_ptr)(rhv);
}
```

Finally, an illustration is provided by the following `main()` function.

- First, A `SortVector` object is created for `MultiData` objects, using the `copy()` generic algorithm to fill the `SortVector` object from information appearing at the program's standard input stream. Having initialized the object its elements are displayed to the standard output stream:

```
SortVector<MultiData> sv;

copy(istream_iterator<MultiData>(cin),
        istream_iterator<MultiData>(),
        back_inserter(sv));
```

- An array of pointers to members is initialized with the addresses of two member functions. The text comparison is considered the most significant sort criterion:

```
bool (MultiData::*arr[])(MultiData const &rhv) const =
{
    &MultiData::textCmp,
    &MultiData::intCmp,
};
```

- Next, the array's elements are sorted and displayed to the standard output stream:

```
sv.hierarchicalSort(arr, 2);
```

- Then the two elements of the array of pointers to `MultiData`'s member functions are swapped, and the previous step is repeated:

```
swap(arr[0], arr[1]);
sv.hierarchicalSort(arr, 2);
```

After compilation the program the following command can be given:

```
echo a 1 b 2 a 2 b 1 | a.out
```

This results in the following output:

```
a 1 b 2 a 2 b 1
====
a 1 a 2 b 1 b 2
====
a 1 b 1 a 2 b 2
====
```

21.10.3 Deriving class templates from ordinary classes

An existing class may be used as the base class for deriving a template class. The advantage of such an inheritance tree is that the base class's members may all be compiled beforehand, so when objects of the class template are instantiated only the used members of the derived (template) class need to be instantiated.

This approach may be used for all class templates having member functions whose implementations do not depend on template parameters. These members may be defined in a separate class which is then used as a base class of the class template derived from it.

As an illustration of this approach we'll develop such a class template in this section. We'll develop a class `Table` derived from an ordinary class `TableType`. The class `Table` will display elements of some type in a table having a configurable number of columns. The elements are either displayed horizontally (the first k elements occupying the first row) or vertically (the first r elements occupying a first column).

When displaying the table's elements they are inserted into a stream. This allows us to define the handling of the table in a separate class (`TableType`), implementing the table's presentation. Since the table's elements are inserted into a stream, the conversion to text (or `string`) can be implemented in `Table`, but the handling of the strings is left to `TableType`. We'll cover some characteristics of `TableType` shortly, concentrating on `Table`'s interface first:

- The class `Table` is a class template, requiring only one template type parameter: `Iterator` refers to an iterator to some data type:

  ```
  template <typename Iterator>
  class Table: public TableType
  {
  ```

- It requires no data members: all data manipulations are performed by `TableType`.
- It has two constructors. The constructor's first two parameters are `Iterator`s used to iterate over the elements to enter into the table. Furthermore, the constructors require us to specify the number of columns we would like our table to have, as well as a *FillDirection*. `FillDirection` is an `enum` type that is actually defined by `TableType`, having values `Horizontal` and `Vertical`. To allow `Table`'s users to exercise control over headers, footers, captions, horizontal and vertical separators, one constructor has `TableSupport` reference parameter. The class `TableSupport` will be developed later as a virtual class allowing clients to exercise this control. Here are the class's constructors:

  ```
  Table(Iterator const &begin, Iterator const &end,
          size_t nColumns, FillDirection direction);
  Table(Iterator const &begin, Iterator const &end,
          TableSupport &tableSupport,
          size_t nColumns, FillDirection direction);
  ```

- The constructors are `Table`'s only two public members. Both constructors use a base class initializer to initialize their `TableType` base class and then call the class's private member `fill()` to insert data into the `TableType` base class object. Here are the constructor's implementations:

```
template <typename Iterator>
Table<Iterator>::Table(Iterator const &begin, Iterator const &end,
                TableSupport &tableSupport,
                size_t nColumns, FillDirection direction)
:
    TableType(tableSupport, nColumns, direction)
{
    fill(begin, end);
}

template <typename Iterator>
Table<Iterator>::Table(Iterator const &begin, Iterator const &end,
                size_t nColumns, FillDirection direction)
:
    TableType(nColumns, direction)
{
    fill(begin, end);
}
```

- The class's `fill()` member iterates over the range of elements `[begin, end)`, as defined by the constructor's first two parameters. As we will see shortly, `TableType` defines a protected data member `std::vector<std::string> d_string`. One of the requirements of the data type to which the iterators point is that this data type can be inserted into streams. So, `fill()` uses a `ostringstream` object to obtain the textual representation of the data, which is then appended to `d_string`:

```
template <typename Iterator>
void Table<Iterator>::fill(Iterator it, Iterator const &end)
{
    while (it != end)
    {
        std::ostringstream str;
        str << *it++;
        d_string.push_back(str.str());
    }
    init();
}
```

This completes the implementation of the class `Table`. Note that this class template only has three members, two of them constructors. Therefore, in most cases only two function templates will have to be instantiated: a constructor and the class's `fill()` member. For example, the following constructs a table having four columns, vertically filled by `string`s extracted from the standard input stream:

```
Table<istream_iterator<string> >
    table(istream_iterator<string>(cin), istream_iterator<string>(),
          4, TableType::Vertical);
```

Note here that the fill-direction is specified as `TableType::Vertical`. It could also have been specified using `Table`, but since `Table` is a class template, the specification would become somewhat more complex: `Table<istream_iterator<string> >::Vertical`.

Now that the `Table` derived class has been designed, let's turn our attention to the class `TableType`. Here are its essential characteristics:

- It is a ordinary class, designed to operate as `Table`'s base class.

- It uses various private data members, among which `d_colWidth`, a vector storing the width of the widest element per column and `d_indexFun`, pointing to the class's member function returning the element in `table[row][column]`, conditional to the table's fill direction. `TableType` also uses a `TableSupport` pointer and a reference. The constructor not requiring a `TableSupport` object uses the `TableSupport *` to allocate a (default) `TableSupport` object and then uses the `TableSupport &` as the object's alias. The other constructor initializes the pointer to 0, and uses the reference data member to refer to the `TableSupport` object provided by its parameter. Alternatively, a static `TableSupport` object might have been used to initialize the reference data member in the former constructor. The remaining private data members are probably self-explanatory:

```
TableSupport           *d_tableSupportPtr;
TableSupport           &d_tableSupport;
size_t                 d_maxWidth;
size_t                 d_nRows;
size_t                 d_nColumns;
WidthType                d_widthType;
std::vector<size_t>    d_colWidth;
size_t                (TableType::*d_widthFun)
                                  (size_t col) const;
std::string const      &(TableType::*d_indexFun)
                                  (size_t row, size_t col) const;
```

- The actual `string` objects populating the table are stored in a protected data member:

```
std::vector<std::string> d_string;
```

- The (protected) constructors perform basic tasks: they initialize the object's data members. Here is the constructor expecting a reference to a `TableSupport` object:

```
#include "tabletype.ih"

TableType::TableType(TableSupport &tableSupport, size_t nColumns,
                        FillDirection direction)
:
    d_tableSupportPtr(0),
    d_tableSupport(tableSupport),
    d_maxWidth(0),
    d_nRows(0),
    d_nColumns(nColumns),
    d_widthType(ColumnWidth),
    d_colWidth(nColumns),
    d_widthFun(&TableType::columnWidth),
    d_indexFun(direction == Horizontal ?
                    &TableType::hIndex
                :
                    &TableType::vIndex)
{}
```

- Once `d_string` has been filled, the table is initialized by `Table::fill()`. The `init()` protected member resizes `d_string` so that its size is exactly `rows x columns`, and it determines the maximum width of the elements per column. Its implementation is straightforward:

```
#include "tabletype.ih"

void TableType::init()
{
    if (!d_string.size())                   // no elements
        return;                             // then do nothing
```

```
    d_nRows = (d_string.size() + d_nColumns - 1) / d_nColumns;
    d_string.resize(d_nRows * d_nColumns);  // enforce complete table

                                            // determine max width per column,
                                            // and max column width
    for (size_t col = 0; col < d_nColumns; col++)
    {
        size_t width = 0;
        for (size_t row = 0; row < d_nRows; row++)
        {
            size_t len = stringAt(row, col).length();
            if (width < len)
                width = len;
        }
        d_colWidth[col] = width;

        if (d_maxWidth < width)             // max. width so far.
            d_maxWidth = width;
    }
}
```

- The public member `insert()` is used by the insertion operator (`operator<<`) to insert a `Table` into a stream. First it informs the `TableSupport` object about the table's dimensions. Next it displays the table, allowing the `TableSupport` object to write headers, footers and separators:

```
#include "tabletype.ih"

ostream &TableType::insert(ostream &ostr) const
{
    if (!d_nRows)
        return ostr;

    d_tableSupport.setParam(ostr, d_nRows, d_colWidth,
                            d_widthType == EqualWidth ? d_maxWidth : 0);

    for (size_t row = 0; row < d_nRows; row++)
    {
        d_tableSupport.hline(row);

        for (size_t col = 0; col < d_nColumns; col++)
        {
            size_t colwidth = width(col);

            d_tableSupport.vline(col);
            ostr << setw(colwidth) << stringAt(row, col);
        }

        d_tableSupport.vline();
    }
    d_tableSupport.hline();

    return ostr;
}
```

- The `cplusplus.yo.zip` archive contains `TableSupport`'s full implementation. This implementation is found in the directory `yo/classtemplates/examples/table`. Most of its remaining members are private. Among those, the following two members return table element `[row][column]` for, respectively, a horizontally filled table and a vertically filled table:

```
    inline std::string const &TableType::hIndex(size_t row, size_t col) const
```

```
{
    return d_string[row * d_nColumns + col];
}
inline std::string const &TableType::vIndex(size_t row, size_t col) const
{
    return d_string[col * d_nRows + row];
}
```

The support class `TableSupport` is used to display headers, footers, captions and separators. It has four virtual members to perform those tasks (and, of course, a virtual constructor):

- `hline(size_t rowIndex)`: called just before the elements in row `rowIndex` will be displayed.
- `hline()`: called immediately after displaying the final row.
- `vline(size_t colIndex)`: called just before the element in column `colIndex` will be displayed.
- `vline()`: called immediately after displaying all elements in a row.

The reader is referrred to the `cplusplus.yo.zip` archive for the full implementation of the classes `Table`, `TableType` and `TableSupport`. Here is a small program showing their use:

```
/*
                                table.cc
*/

#include <fstream>
#include <iostream>
#include <string>
#include <iterator>
#include <sstream>

#include "tablesupport/tablesupport.h"
#include "table/table.h"

using namespace std;
using namespace FBB;

int main(int argc, char **argv)
{
    size_t nCols = 5;
    if (argc > 1)
    {
        istringstream iss(argv[1]);
        iss >> nCols;
    }

    istream_iterator<string>   iter(cin);   // first iterator isn't const

    Table<istream_iterator<string> >
        table(iter, istream_iterator<string>(), nCols,
              argc == 2 ? TableType::Vertical : TableType::Horizontal);

    cout << table << endl;
    return 0;
}
/*
    Example of generated output:
```

```
        After: echo a b c d e f g h i j | demo 3
            a e i
            b f j
            c g
            d h
        After: echo a b c d e f g h i j | demo 3 h
            a b c
            d e f
            g h i
            j
    */
```

21.11 Class templates and nesting

When a class is nested within a class template, it automatically becomes a class template itself. The nested class may use the template parameters of the surrounding class, as shown in the following skeleton program. Within a class `PtrVector`, a class `iterator` is defined. The nested class receives its information from its surrounding class, a `PtrVector<Type>` class. Since this surrounding class should be the only class constructing its iterators, `iterator`'s constructor is made private, and the surrounding class is given access to the private members of `iterator` using a *bound friend* declaration. Here is the initial section of `PtrVector`'s class interface:

```
template <typename Type>
class PtrVector: public std::vector<Type *>
```

This shows that the `std::vector` base class will store pointers to `Type` values, rather than the values themselves. Of course, a destructor is needed now, since the (externally allocated) memory for the `Type` objects must eventually be freed. Alternatively, the allocation might be part of `PtrVector`'s tasks, when storing new elements. Here it is assumed that the `PtrVector`'s clients do the required allocations, and that the destructor will be implemented later on.

The nested class defines its constructor as a private member, and allows `PtrVector<Type>` objects to access its private members. Therefore only objects of the surrounding `PtrVector<Type>` class type are allowed to construct their `iterator` objects. However, `PtrVector<Type>`'s clients may construct *copies* of the `PtrVector<Type>::iterator` objects they use. Here is the nested class `iterator`, containing the required `friend` declaration. Note the use of the `typename` keyword: since `std::vector<Type *>::iterator` depends on a template parameter, it is not yet an instantiated class, so `iterator` becomes an implicit typename. The compiler issues a corresponding warning if `typename` has been omitted. In these cases `typename` must be used. Here is the class interface:

```
        class iterator
        {
            friend class PtrVector<Type>;
            typename std::vector<Type *>::iterator d_begin;

            iterator(PtrVector<Type> &vector);

            public:
                Type &operator*();
        };
```

The implementation of the members shows that the base class's `begin()` member is called to initialize `d_begin`. Also note that the return type of `PtrVector<Type>::begin()` must again be preceded by `typename`:

```
template <typename Type>
```

```
PtrVector<Type>::iterator::iterator(PtrVector<Type> &vector)
:
    d_begin(vector.std::vector<Type *>::begin())
{}

template <typename Type>
Type &PtrVector<Type>::iterator::operator*()
{
    return **d_begin;
}
```

The remainder of the class is simple. Omitting all other functions that might be implemented, the function `begin()` will return a newly constructed `PtrVector<Type>::iterator` object. It may call the constructor since the class `iterator` called its surrounding class its friend:

```
template <typename Type>
typename PtrVector<Type>::iterator PtrVector<Type>::begin()
{
    return iterator(*this);
}
```

Here is a simple skeleton program, showing how the nested class `iterator` might be used:

```
int main()
{
    PtrVector<int> vi;

    vi.push_back(new int(1234));

    PtrVector<int>::iterator begin = vi.begin();

    std::cout << *begin << endl;
}
```

Nested enumerations and typedefs can also be defined in class templates. The class `Table`, mentioned before (section 21.10.3) inherited the enumeration `TableType::FillDirection`. If `Table` would have been implemented as a full class template, then this enumeration would have been defined in `Table` itself as:

```
template <typename Iterator>
class Table: public TableType
{
    public:
        enum FillDirection
        {
            Horizontal,
            Vertical
        };
    ...
};
```

In this case, the actual value of the template type parameter must be specified when referring to a `FillDirection` value or to its type. For example (assuming `iter` and `nCols` are defined as in section 21.10.3):

```
Table<istream_iterator<string> >::FillDirection direction =
            argc == 2 ?
```

```
                Table<istream_iterator<string> >::Vertical
            :
                Table<istream_iterator<string> >::Horizontal;

Table<istream_iterator<string> >
    table(iter, istream_iterator<string>(), nCols, direction);
```

21.12 Constructing iterators

In section 18.2 the iterators used with generic algorithms were introduced. We've seen that several types of iterators were distinguished: InputIterators, ForwardIterators, OutputIterators, BidirectionalIterators and RandomAccessIterators.

In section 18.2 the characteristics of iterators were introduced: all iterators should support an increment operation, a dereference operation and a comparison for (in)equality.

However, when iterators must be used in the context of generic algorithms they must meet additional requirements. This is caused by the fact that generic algorithms check the types of the iterators they receive. Simple pointers are usually accepted, but if an iterator-object is used it must be able to specify the kind of iterator it represents.

To ensure that an object of a class is interpreted as a particular type of iterator, the class must be derived from the `class iterator`. The particular type of iterator is defined by the class template's *first* parameter, and the particular data type to which the iterator points is defined by the class template's *second* parameter. Before a class may be inherited from the class `iterator`, the following header file must have been included:

```
#include <iterator>
```

The particular type of iterator that is implemented by the derived class is specified using a so-called *iterator_tag*, provided as the first template argument of the class `iterator`. For the five basic iterator types, these tags are:

- `std::input_iterator_tag`. This tag defines an InputIterator. Iterators of this type allow reading operations, iterating from the first to the last element of the series to which the iterator refers.
- `std::output_iterator_tag`. This tag defines an OutputIterator. Iterators of this type allow for assignment operations, iterating from the first to the last element of the series to which the iterator refers.
- `std::forward_iterator_tag`. This tag defines a `ForwardIterator`. Iterators of this type allow reading *and* assignment operations, iterating from the first to the last element of the series to which the iterator refers.
- `std::bidirectional_iterator_tag`. This tag defines a BidirectionalIterator. Iterators of this type allow reading *and* assignment operations, iterating step by step, possibly in alternating directions, over all elements of the series to which the iterator refers.
- `std::random_access_iterator_tag`. This tag defines a RandomAccessIterator. Iterators of this type allow reading *and* assignment operations, iterating, possibly in alternating directions, over all elements of the series to which the iterator refers using any available (random) stepsize.

Each *iterator tag* assumes that a certain set of operators is available. The *RandomAccessIterator* is the most complex of iterators, as it implies all other iterators.

Note that iterators are always defined over a certain range, e.g., `[begin, end)`. Increment and decrement operations may result in undefined behavior of the iterator if the resulting iterator value would refer to a location outside of this range.

Often, iterators only access the elements of the series to which they refer. Internally, an iterator may use an ordinary pointer, but it is hardly ever necessary for the iterator to allocate its own memory. Therefore, as the overloaded assignment operator and the copy constructor do not have to allocate any memory, the *default implementation* of the overloaded assignment operator and of the copy constructor is usually sufficient. I.e., usually these members do not have to be implemented at all. As a consequence there is usually also no *destructor*.

Most classes offering members returning iterators do so by having members constructing the required iterator, which is thereupon returned as an object by these member functions. As the *caller* of these member functions only has to *use* or sometimes *copy* the returned iterator objects, there is normally no need to provide any publicly available constructors, except for the copy constructor. Therefore these constructors may usually be defined as *private* or *protected* members. To allow an outer class to create iterator objects, the iterator class will declare the outer class as its *friend*.

In the following sections, the construction of a *RandomAccessIterator*, the most complex of all iterators, and the construction of a *reverse RandomAccessIterator* is discussed. The container class for which a random access iterator must be developed may actually store its data elements in many different ways, e.g., using various containers or using pointers to pointers. Therefore it is difficult to construct a template iterator class which is suitable for a large variety of ordinary (container) classes.

In the following sections, the available `std::iterator` class will be used to construct an inner class representing a random access iterator. This approach clearly shows how to construct an iterator class. The reader may either follow this approach when constructing iterator classes in other contexts, or a full template iterator class can be designed. An example of such a template iterator class is provided in section 23.6.

The construction of the random access iterator as shown in the next sections aims at the implementation of an iterator reaching the elements of a series of elements only accessible through pointers. The iterator class is designed as an inner class of a class derived from a vector of string pointers.

21.12.1 Implementing a 'RandomAccessIterator'

When discussing containers (chapter 12) it was noted that containers own the information they contain. If they contain objects, then these objects are destroyed once the containers are destroyed. As pointers are no objects their use in containers is discouraged (STL's `unique_ptr` and `shared_ptr` type objects may be used, though). Although discouraged, we might be able to use pointer data types in specific contexts. In the following class `StringPtr`, a ordinary class is derived from the `std::vector` container using `std::string *` as its data type:

```
#ifndef INCLUDED_STRINGPTR_H_
#define INCLUDED_STRINGPTR_H_

#include <string>
#include <vector>

class StringPtr: public std::vector<std::string *>
{
    public:
        StringPtr(StringPtr const &other);
        ~StringPtr();

        StringPtr &operator=(StringPtr const &other);
};

#endif
```

Note the declaration of the destructor: as the object stores string pointers, a destructor is required to destroy the strings when the `StringPtr` object itself is destroyed. Similarly, a copy constructor

and overloaded assignment is required. Other members (in particular: constructors) are not explicitly declared as they are not relevant to this section's topic.

Let's assume that we want to be able to use the `sort()` generic algorithm with `StringPtr` objects. This algorithm (see section 19.1.58) requires two *RandomAccessIterators*. Although these iterators are available (via `std::vector`'s `begin()` and `end()` members), they return iterators to `std::string *`s, which cannot sensibly be compared.

To remedy this, assume that we have defined an internal type `StringPtr::iterator`, not returning iterators to pointers, but iterators to the objects these pointers point to. Once this `iterator` type is available, we can add the following members to our `StringPtr` class interface, hiding the identically named, but useless members of its base class:

```
StringPtr::iterator begin();    // returns iterator to the first element
StringPtr::iterator end();      // returns iterator beyond the last
                                // element
```

Since these two members return the (proper) iterators, the elements in a `StringPtr` object can easily be sorted:

```
in main()
{
    StringPtr sp;               // assume sp is somehow filled

    sort(sp.begin(), sp.end()); // sp is now sorted
    return 0;
}
```

To make this all work, the type `StringPtr::iterator` must be defined. As suggested by its type name, `iterator` is a nested type of `StringPtr`, suggesting that we may implement `iterator` as a nested class of `StringPtr`. However, to use a `StringPtr::iterator` in combination with the `sort()` generic algorithm, it must also be a `RandomAccessIterator`. Therefore, `StringPtr::iterator` itself must be derived from the existing class `std::iterator`, available once the following preprocessor directive has been specified:

```
    #include <iterator>
```

To derive a class from `std::iterator`, both the iterator type and the data type the iterator points to must be specified. Take caution: our iterator will take care of the `string *` dereferencing; so the required data type will be `std::string`, and *not* `std::string *`. So, the class `iterator` starts its interface as:

```
class iterator:
    public std::iterator<std::random_access_iterator_tag, std::string>
```

Since its base class specification is quite complex, we could consider associating this type with a shorter name using the following `typedef`:

```
typedef std::iterator<std::random_access_iterator_tag, std::string>
        Iterator;
```

However, if the defined type (`Iterator`) is used only once or twice, the typedefinition only adds clutter to the interface, and is better not used.

Now we're ready to redesign `StringPtr`'s class interface. It contains members returning (reverse) iterators, and a nested `iterator` class. The members will be discussed in some detail next:

```
class StringPtr: public std::vector<std::string *>
```

```
{
    public:
    class iterator: public
            std::iterator<std::random_access_iterator_tag, std::string>
    {
        friend class StringPtr;
        std::vector<std::string *>::iterator d_current;

        iterator(std::vector<std::string *>::iterator const &current);

        public:
            iterator &operator--();
            iterator const operator--(int);
            iterator &operator++();
            bool operator==(iterator const &other) const;
            bool operator!=(iterator const &other) const;
            int operator-(iterator const &rhs) const;
            std::string &operator*() const;
            bool operator<(iterator const &other) const;
            iterator const operator+(int step) const;
            iterator const operator-(int step) const;
            iterator &operator+=(int step); // increment over 'n' steps
            iterator &operator-=(int step); // decrement over 'n' steps
            std::string *operator->() const;// access the fields of the
                                            // struct an iterator points
                                            // to. E.g., it->length()
    };

    typedef std::reverse_iterator<iterator> reverse_iterator;

    iterator begin();
    iterator end();
    reverse_iterator rbegin();
    reverse_iterator rend();
};
```

Let's first have a look at `StringPtr::iterator`'s characteristics:

- `iterator` defines `StringPtr` as its friend, so `iterator`'s constructor can remain private: only the `StringPtr` class itself is now able to construct `iterator`s, which seems like a sensible thing to do. Under the current implementation, *copy-constructing* remains of course possible. Furthermore, since an iterator is already provided by `StringPtr`'s base class, we can use that iterator to access the information stored in the `StringPtr` object.

- `StringPtr::begin()` and `StringPtr::end()` may simply return `iterator` objects. Their implementations are:

```
inline StringPtr::iterator StringPtr::begin()
{
    return iterator(this->std::vector<std::string *>::begin());
}
inline StringPtr::iterator StringPtr::end()
{
    return iterator(this->std::vector<std::string *>::end());
}
```

- All of `iterator`'s remaining members are public. It's very easy to implement them, mainly manipulating and dereferencing the available iterator `d_current`. A `RandomAccessIterator`

(which is the most complex of iterators) requires a series of operators. They usually have very simple implementations, making them good candidates for inline-members:

- `iterator &operator++()`: the pre-increment operator:

```
inline StringPtr::iterator &StringPtr::iterator::operator++()
{
    ++d_current;
    return *this;
}
```

- `iterator &operator--`: the pre-decrement operator:

```
inline StringPtr::iterator &StringPtr::iterator::operator--()
{
    --d_current;
    return *this;
}
```

- `iterator operator--`: the post-decrement operator:

```
inline StringPtr::iterator const StringPtr::iterator::operator--(int)
{
    return iterator(d_current--);
}
```

- `iterator &operator=(iterator const &other)`: the overloaded assignment operator. Since `iterator` objects do not allocate any memory themselves, the default assignment operator will do.
- `bool operator==(iterator const &rhv) const`: testing the equality of two `iterator` objects:

```
inline bool StringPtr::iterator::operator==(iterator const &other) const
{
    return d_current == other.d_current;
}
```

- `bool operator<(iterator const &rhv) const`: tests whether the left-hand side iterator points to an element of the series located *before* the element pointed to by the right-hand side iterator:

```
inline bool StringPtr::iterator::operator<(iterator const &other) const
{
    return **d_current < **other.d_current;
}
```

- `int operator-(iterator const &rhv) const`: returns the number of elements between the element pointed to by the left-hand side iterator and the right-hand side iterator (i.e., the value to add to the left-hand side iterator to make it equal to the value of the right-hand side iterator):

```
inline int StringPtr::iterator::operator-(iterator const &rhs) const
{
    return d_current - rhs.d_current;
}
```

- `Type &operator*() const`: returns a reference to the object to which the current iterator points. With an `InputIterator` and with all `const_iterators`, the return type of this overloaded operator should be `Type const &`. This operator returns a reference to a string. This string is obtained by dereferencing the dereferenced `d_current` value. As `d_current` is an iterator to `string *` elements, two dereference operations are required to reach the string itself:

```
inline std::string &StringPtr::iterator::operator*() const
{
    return **d_current;
}
```

- `iterator const operator+(int stepsize) const`: this operator advances the current iterator by `stepsize` steps:

```
inline StringPtr::iterator const
        StringPtr::iterator::operator+(int step) const
{
    return iterator(d_current + step);
}
```

- `iterator const operator-(int stepsize) const`: this operator decreases the current iterator by `stepsize` steps:

```
inline StringPtr::iterator const
       StringPtr::iterator::operator-(int step) const
{
    return iterator(d_current - step);
}
```

- iterators may be constructed from existing iterators. This constructor doesn't have to be implemented, as the default copy constructor can be used.
- `std::string *operator->() const` is an additionally added operator. Here only one dereference operation is required, returning a pointer to the string, allowing us to access the members of a string via its pointer.

```
inline std::string *StringPtr::iterator::operator->() const
{
    return *d_current;
}
```

- Two more additionally added operators are `operator+=()` and `operator-=()`. They are not formally required by `RandomAccessIterators`, but they come in handy anyway:

```
inline StringPtr::iterator &StringPtr::iterator::operator+=(int step)
{
    d_current += step;
    return *this;
}
inline StringPtr::iterator &StringPtr::iterator::operator-=(int step)
{
    d_current -= step;
    return *this;
}
```

The interfaces required for other iterator types are simpler, requiring only a subset of the interface required by a random access iterator. E.g., the forward iterator is never decremented and never incremented over arbitrary step sizes. Consequently, in that case all decrement operators and `operator+(int step)` can be omitted from the interface. Of course, the tag to use would then be `std::forward_iterator_tag`. The tags (and the set of required operators) varies accordingly for the other iterator types.

21.12.2 Implementing a 'reverse_iterator'

Once we've implemented an iterator, the matching *reverse iterator* can be implemented in a jiffy. Comparable to the `std::iterator` a `std::reverse_iterator` exists, which will nicely implement the reverse iterator for us, once we have defined an iterator class. Its constructor merely requires an object of the iterator type for which we want to construct a reverse iterator.

To implement a reverse iterator for `StringPtr`, we only need to define the `reverse_iterator` type in its interface. This requires us to specify only one line of code, which must be inserted after the interface of the class `iterator`:

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Finally, the well known members `rbegin()` and `rend()` are added to `StringPtr`'s interface. Again, they can easily be implemented inline:

```
inline StringPtr::reverse_iterator StringPtr::rbegin()
{
    return reverse_iterator(end());
}
inline StringPtr::reverse_iterator StringPtr::rend()
{
    return reverse_iterator(begin());
}
```

Note the arguments the `reverse_iterator` constructors receive: the *begin point* of the reversed iterator is obtained by providing `reverse_iterator`'s constructor with `end()`: the *endpoint* of the normal iterator range; the *endpoint* of the reversed iterator is obtained by providing `reverse_iterator`'s constructor with `begin()`: the *begin point* of the normal iterator range.

The following little program illustrates the use of `StringPtr`'s `RandomAccessIterator`:

```
    #include <iostream>
    #include <algorithm>
    #include "stringptr.h"
    using namespace std;

    int main(int argc, char **argv)
    {
        StringPtr sp;

        while (*argv)
            sp.push_back(new string(*argv++));

        sort(sp.begin(), sp.end());
        copy(sp.begin(), sp.end(), ostream_iterator<string>(cout, " "));

        cout << "\n======\n";

        sort(sp.rbegin(), sp.rend());
        copy(sp.begin(), sp.end(), ostream_iterator<string>(cout, " "));

        cout << endl;
    }
    /*
            when called as:
        a.out bravo mike charlie zulu quebec

            generated output:
        a.out bravo charlie mike quebec zulu
        ======
        zulu quebec mike charlie bravo a.out
    */
```

Although it is thus possible to construct a reverse iterator from a normal iterator, the opposite does not hold true: it is not possible to initialize a normal iterator from a reverse iterator. Let's assume we would like to process all lines stored in a `vector<string> lines` up to any trailing empty lines (or lines only containing blanks) it might contain. How would we proceed? One approach is to start the processing from the first line in the vector, continuing until the first of the trailing empty lines. However, once we encounter an empty line it does of course not have to be the first line of the set of trailing empty lines. In that case, we would like to use the following algorithm:

- First, use

```
rit = find_if(lines.rbegin(), lines.rend(), NonEmpty());
```

  to obtain a `reverse_iterator rit` pointing to the last non-empty line.

- Next, use

```
for_each(lines.begin(), --rit, Process());
```

  to process all lines up to the first empty line.

However, we can't mix iterators and reverse iterators when using generic algorithms. So how can we initialize the second iterator using the available `reverse_iterator`? The solution is actually not very difficult, as an iterator may be initialized by a pointer. The reverse iterator `rit` is not a pointer, but `&*(rit - 1)` or `&*-rit` is. Thus, we can use

```
for_each(lines.begin(), &*--rit, Process());
```

to process all the lines up to the first of the set of trailing empty lines. In general, if `rit` is a `reverse_iterator` pointing to some element, but we need an `iterator` to point to that element, we may use `&*rit` to initialize the iterator. Here, the dereference operator is applied to reach the element the reverse iterator refers to. Then the address operator is applied to obtain its address.

Chapter 22

Advanced Template Use

The main purpose of templates is to provide a generic definition of classes and functions which can then be tailored to specific types when required.

However, templates allow us to do more than that. If not for compiler implementation limitations, templates could be used to program, compile-time, just about anything we use computers for. This remarkable feat, offered by no other current day computer language, stems from the fact that templates allow us to do three things compile-time:

- Templates allow us to to integer arithmetic and to save computed values symbolically;
- Templates allow us to make compile-time decisions;
- Templates allow us to do things repeatedly

Of course, asking the compiler to compute, e.g., prime numbers, is one thing. It's a completely different thing to do so in an award winning way. Don't expect speed records to be broken when the compiler performs complex calculations for us. But that's al beside the point, which is that we *can* ask the compiler to compute virtually anything using **C++**'s template language.

In this chapter these remarkable features of templates are discussed. Following a short overview of subtleties related to templates the main characteristics of template meta programming are introduced.

Following that discussion a third type of template parameter, the *template template parameter* is introduced, laying the groundwork for the discusion of *trait classes* and *policy classes*.

This chapter ends with the discussion of several additional and interesting applications of templates: adapting compiler error messages, conversions to class types and an elaborate example discussing compile-time list processing.

Much of the inspiration for this chapter resulted from two highly recommended books: Andrei Alexandrescu's 2001 book **Modern C++ design** (Addison-Wesley) and Nicolai Josutis and David Vandevoorde's 2003 book **Templates** (Addison-Wesley)

22.1 Subtleties

In this section the following topics are covered:

- In section 22.1.1 the use of the keyword `typename` is discussed. It is used to distinguish types defined by class templates from members defined by class templates;
- in section 22.1.2 applies `typename` to situations where types nested in templates are returned from member functions of class templates;

- in section 22.1.3 covers the problem of how to refer to base class templates from derived class templates;
- and section 22.1.4 covers some new syntax: `::template` and variants, used to inform the compiler that a name used inside a template is itself a class template.

22.1.1 The keyword 'typename'

The keyword `typename` has been used until now to indicate a template type parameter. However, it is also used to disambiguate code inside templates. Consider the following code:

```
template <typename Type>
Type function(Type t)
{
    Type::Ambiguous *ptr;

    return t + *ptr;
}
```

When this code is shown to the compiler, it will complain with an -at first sight puzzling- error message like:

```
demo.cc:4: error: 'ptr' was not declared in this scope
```

The puzzling nature of this error message is that the intention of the programmer was actually to declare a pointer to a type `Ambiguous` defined within the class template `Type`. However, the compiler, when confronted with `Type::Ambiguous` has to make a decision about the nature of this construct. Clearly it cannot inspect `Type` to find out its true nature, since `Type` is a template type, and hence its actual definition isn't available yet. The compiler now is confronted with two possibilities: either `Type::Ambiguous` is a *static member* of the as yet mysterious template `Type`, or it is a *subtype* defined by `Type`. As the standard specifies that the compiler must assume the former, the statement

```
Type::Ambiguous *ptr;
```

is eventually interpreted as a *multiplication* of the static member `Type::Ambiguous` and the (now undeclared) entity `ptr`. The reason for the error message should now be clear: in this context `ptr` is unknown.

To disambiguate code in which an identifier refers to a type that is itself a subtype of a template type parameter the keyword `typename` must be used. Accordingly, the above code is altered into:

```
template <typename Type>
Type function(Type t)
{
    typename Type::Ambiguous *ptr;

    return t + *ptr;
}
```

Classes fairly often define subtypes. When such classes are thought of when designing templates, these subtypes may appear inside the template's definitions as subtypes of template type parameters, requiring the use of the `template` keyword. E.g., assume a class template `Handler` defines a `typename Container` as its type parameter, as well as a data member storing the container's `begin()` iterator. Furthermore, the class template `Handler` may offer a constructor accepting any container supporting a `begin()` member. The skeleton of the class `Handler` could then be:

```
template <typename Container>
```

```
class Handler
{
    Container::const_iterator d_it;

    public:
        Handler(Container const &container)
        :
            d_it(container.begin())
        {}
};
```

What were the considerations we had in mind when designing this class?

- The typename `Container` represents any container supporting iterators.
- The container presumably supports a member `begin()`. The initialization `d_it(container.begin())` clearly depends on the template's type parameter, so it's only checked for basic syntactic correctness.
- Likewise, the container presumably supports a *subtype* `const_iterator`, defined in the class `Container`.

The final consideration is an indication that `typename` is required. If this is omitted, and a `Handler` is instantiated because of the following `main()` function one again a peculiar compilation error is generated:

```
#include "handler.h"
#include <vector>
using namespace std;

int main()
{
    vector<int> vi;
    Handler<vector<int> > ph(vi);
}
/*
    Reported error:

handler.h:4: error: syntax error before ';' token
*/
```

Clearly the line

```
    Container::const_iterator d_it;
```

in the `Handler` class causes a problem: it is interpreted by the compiler as a *static member* instead of a subtype. Again, the problem is solved using `typename`:

```
template <typename Container>
class Handler
{
    typename Container::const_iterator d_it;
    ...
};
```

An interesting illustration that the compiler indeed assumes `X::a` to be a member `a` of the class `X` is provided by the error message we get when we try to compile `main()` using the following implementation of `Handler`'s constructor:

```
    Handler(Container const &container)
    :
        d_it(container.begin())
    {
        size_t x = Container::ios_end;
    }
    /*
        Reported error:

        error: 'ios_end' is not a member of type 'std::vector<int,
                std::allocator<int> >'
    */
```

As a final illustration consider what happens if the function template introduced at the beginning of this section doesn't return a `Type` value, but a `Type::Ambiguous` value. Again, a subtype of a template type is referred to, and `typename` is required:

```
    template <typename Type>
    typename Type::Ambiguous function(Type t)
    {
        return t.ambiguous();
    }
```

Using `typename` in the specification of a return type is further discussed in section 22.1.2.

In some cases `typename` can be avoided by resorting to a `typedef`. E.g., `Iterator`, defined using `typedef`, can be used to indicate the specific type:

```
    template <typename Container>
    class Handler
    {
        typedef typename Container::const_iterator Iterator;

        Iterator d_it;
        ...
    };
```

22.1.2 Returning types nested under class templates

Consider the following example in which a nested class, that is not depending on a template parameter, is defined within a template class. Furthermore, the class template member `nested()` returns an object of the nested class. Note that a (deprecated) in-class member implementation is used. The reason for this will become clear shortly.

```
    template <typename T>
    class Outer
    {
        public:
            class Nested
            {};

            Nested nested() const
            {
                return Nested();
            }
    };
```

The above example compiles flawlessly: within the class `Outer` there is no ambiguity with respect to the meaning of `nested()`'s return type.

However, since it is advised to implement inline and template members below their class interface (see section 7.6.1), we now remove the implementation from the interface itself, and put it below the interface. Suddenly the compiler refuses to compile our member `nested()`:

```
template <typename T>
class Outer
{
    public:
        class Nested
        {};

        Nested nested() const;
};

template <typename T>
Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}
```

The above implementation of `nested()` produces an error message like

error: expected constructor, destructor, or type conversion before 'Outer'.

In cases like these the return type (i.e., `Outer<T>::Nested`) refers to a *subtype* of `Outer<T>` rather than to a member of `Outer<T>`.

As a general rule the following holds true: the keyword `typename` must be used whenever a type is referred to that is a *subtype* of a type that is itself depending on a template type parameter. Writing `typename` in front of `Outer<T>::Nested` removes the compilation error and therefore the correct implementation of the function `nested()` becomes:

```
template <typename T>
typename Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}
```

22.1.3 Type resolution for base class members

Consider the following example of a template base and a derived class:

```
#include <iostream>

template <typename T>
class Base
{
    public:
        void member();
};

template <typename T>
void Base<T>::member()
```

```
{
    std::cout << "This is Base<T>::member()\n";
}

template <typename T>
class Derived: public Base<T>
{
    public:
        Derived();
};

template <typename T>
Derived<T>::Derived()
{
    member();
}
```

This example won't compile, and the compiler tells us something like:

```
error: there are no arguments to 'member' that depend on a template
       parameter, so a declaration of 'member' must be available
```

At first glance, this error may cause some confusion, since with non-class templates public and protected base class members are immediately available. This also holds true for class templates, but only if the compiler can figure out what we mean. In the above situation, the compiler can't, since it doesn't know for what type `T` the member function `member` must be initialized.

To appreciate why this is true, consider the situation where we have defined a specialization:

```
template <>
Base<int>::member()
{
    std::cout << "This is the int-specialization\n";
}
```

Since the compiler, when processing the class `Derived`, can't be sure that no specialization will be in effect once an instantiation of `Derived` is called for, it can't decide yet for what type to instantiate `member`, since `member()`'s call in `Derived::Derived()` doesn't require a template type parameter. In cases like these, where no template type parameter is available to determine which type to use, the compiler must be told that it should postpone its decision about the template type parameter to use for `member()` until instantiation time. This can be implemented in two ways: either by using `this`, or by explicitly mentioning the base class, instantiated for the derived class's template type(s). In the following `main()` function both forms are used. Note that with the `int` template type the `int` specialization is used.

```
#include <iostream>

template <typename T>
class Base
{
    public:
        void member();
};

    template <typename T>
    void Base<T>::member()
    {
        std::cout << "This is Base<T>::member()\n";
```

```
    }

    template <>
    void Base<int>::member()
    {
        std::cout << "This is the int-specialization\n";
    }

template <typename T>
class Derived: public Base<T>
{
    public:
        Derived();
};

    template <typename T>
    Derived<T>::Derived()
    {
        this->member();         // Using 'this' implies <T> at
                                // instantiation time.
        Base<T>::member();      // Same.
    }

int main()
{
    Derived<double> d;
    Derived<int> i;
}

/*
    Generated output:
    This is Base<T>::member()
    This is Base<T>::member()
    This is the int-specialization
    This is the int-specialization
*/
```

22.1.4 ::template, .template and ->template

In general, the compiler is able to determine the true nature of a name. As discussed in the previous sections, this is not always the case and the software engineer sometimes has to advise the compiler. The `typename` keyword can often used to that purpose.

However, `typename` cannot always come to the rescue. While parsing a source, the compiler receives a series of *tokens*, representing meaningful units of text encountered in the program's source. A token represents, e.g., an identifier or a number. Other tokens represent operators, like `=`, `+` or `<`. It is precisely the last token that may cause problems, as it is used in multiple ways, which cannot always be determined from the context in which the compiler encounters `<`. Sometimes, however, the compiler *will* know that `<` does not represent the *less than* operator, as in the situation where a template parameter list follows the keyword `template`, e.g.,

```
template <typename T, int N>
```

Clearly, in this case `<` does not represent a 'less than' operator.

The special meaning of `<` if preceded by `template` forms the basis for the syntactic constructs discussed in this section.

Assume the following class has been defined:

```
template <typename Type>
class Outer
{
    public:
        template <typename InType>
        class Inner
        {
            public:
                template <typename X>
                void nested();
        };
};
```

Here a class template `Outer` defines a nested class template `Inner`, which in turn defines a template member function.

Next, a class template `Usage` is defined, offering a member function `caller()` expecting an object of the above `Inner` type. An initial setup for `Usage` could be written as follows:

```
template <typename T1, typename T2>
class Usage
{
    public:
        void fun(Outer<T1>::Inner<T2> &obj);
    ...
};
```

The compiler, however, won't accept this. It interprets `Outer<T1>::Inner` as a class type, which of course doesn't exist. In this situation the compiler generates an error like:

```
error: 'class Outer<T1>::Inner' is not a type
```

To inform the compiler that in this case `Inner` itself is a template, using the template type parameter `<T2>`, the `::template` construction is required. This tells the compiler that the next `<` should not be interpreted as a 'less than' token, but rather as a template type argument. So, the declaration is modified to:

```
void fun(Outer<T1>::template Inner<T2> &obj);
```

But this still doesn't get us where we want to be: after all `Inner<T2>` is a type, nested under a class template, depenting on a template type parameter. Actually, the compiler produces a series of error messages here, one of them being like:

```
error: expected type-name before '&' token
```

which nicely indicates what should be done to get it right: add `typename`:

```
void fun(typename Outer<T1>::template Inner<T2> &obj);
```

Next, `fun()` itself is not only just declared, it is implemented as well. The implementation should call `Inner`'s member `nested()` function, instantiated for yet another type `X`. The class template `Usage` should now receive a third template type parameter, which can be called `T3`: let's assume it has been defined. To implement `fun()`, we start out with:

```
void fun(typename Outer<T1>::template Inner<T2> &obj)
```

```
{
    obj.nested<T3>();
}
```

However, once again we run into a problem. The compiler once again interprets < as 'less than', and expects a logical expression, having as its right-hand side a primary expression instead of a formal template type.

To tell the compiler in situations like these that <T3> should be interpreted as a type to instantiate nested with, the template keyword is used once more. This time it is used in the context of the member selection operator: by writing .template the compiler is informed that what follows is not a 'less than' operator, but rather a type specification. The function's final implementation becomes:

```
void fun(typename Outer<T1>::template Inner<T2> &obj)
{
    obj.template nested<T3>();
}
```

Instead of value or reference parameters functions may define pointer parameters. If obj would have been defined as a pointer parameter the implementation would use the ->template construction, rather than the .template construction. E.g.,

```
void fun(typename Outer<T1>::template Inner<T2> *ptr)
{
    ptr->template nested<T3>();
}
```

22.2 Template Meta Programming

22.2.1 Values according to templates

In template programming values are preferably represented by enum values. Enums are preferred over, e.g., int const values since enums never have any linkage: they are pure symbolic values with no memory representation.

Consider the situation where a programmer must use a cast, say a reinterpret_cast. A problem with a reinterpret_cast is that it is the ultimate way to turn off all compiler checks. All bets are off, and we can write extreme but absolutely pointless reinterpret_cast statements, like

```
int value = 12;
ostream &ostr = reinterpret_cast<ostream &>(value);
```

Wouldn't it be nice if the compiler would warn us against such oddities by generating an error message? If that's what we'd like the compiler to do, there must be some way to distinguish madness from weirdness. Let's assume we agree on the following distinction: reinterpret casts are never acceptable if the target type represents a larger type than the expression (source) type, since that would immediately result in abusing the amount of memory that's actually available to the target type. In this way we can't allow reinterpret cast from int to double since a double is a larger type than an int.

The intent is now to create a new kind of cast, let's call it reinterpret_to_smaller_cast, which can only be performed if the target type is a *smaller* type than the source type (note that this exactly the opposite readoning as used by Alexandrescu (2001), section 2.1).

The following template is constructed:

```
template<typename Target, typename Source>
```

```
Target &reinterpret_to_smaller_cast(Source &source)
{
    // determine whether Target is smaller than source
    return reinterpret_cast<Target &>(source);
}
```

At the comment an enum-definition is inserted with a suggestive name, resulting in a compile-time error if the condition is not met. A division by zero is clearly not allowed, and noting that a `false` value represents a zero value, the condition could be:

```
1 / (sizeof(Target) <= sizeof(Source));
```

The interesting part is that this condition doesn't result in any code at all: it's a mere value that's computed by the compiler while compiling the expression. To transform this into a useful error message the expression is assigned to a desciptive `enum` value, resulting in, e.g.,

```
template<typename Target, typename Source>
Target &reinterpret_to_smaller_cast(Source &source)
{
    enum
    {
        the_Target_size_exceeds_the_Source_size =
            1 / (sizeof(Target) <= sizeof(Source))
    };
    return reinterpret_cast<Target &>(source);
}
```

When `reinterpret_to_smaller_cast` is used to cast from `int` to `ostream` an error is produced by the compiler, like:

```
error: enumerator value for 'the_Target_size_exceeds_the_Source_size'
    not integer constant
```

whereas no error is reported if, e.g., `reinterpret_to_smaller_cast<int>(cout)` is requested.

In the above example a `enum` was used to compute compile time a value that is illegal if an assumption is not met. The creative part is finding an appropriate expression.

Enum values are well suited for these situations as they do not consume any memory and their evaluation does not produce any executable code. They can be used to accumulate values too: the resulting enum value will then contain a final value, computed by the compiler rather than by code as the next sections illustrate. In general, programs shouldn't do run-time what they can do compile-time and computing complex calculations resulting in constant values is a clear example of this principle.

22.2.1.1 Converting integral types to types

Another use of values buried inside templates is to 'templatize' simple scalar `int` values. This is primarily useful in situations where a scalar value (often a `bool` value) is available to select an appropriate member specialization, a situation that will be encountered shortly (section 22.2.2).

Templatizing integral values is based on the fact that a class template together with its template arguments represent a type. E.g., `vector<int>` and `vector<double>` are different types.

Templatizing integral values is simply implemented: just define a template, it does not have to have any contents at all, but it customarily has the integral values stored as an `enum` value:

```
template <int x>
```

```
struct IntType
{
    enum { value = x };
};
```

Since `IntType` does not have any members, but just the `enum` value, the '`class`' can be defined as a '`struct`', saving us from typing `public:`. Defining the `enum` value '`value`' allows us to retrieve the value used at the instantiation at no cost in memory: enum values are not variables or data members, and thus have no address. They are mere values.

Using the `struct IntType` is easy: just define an anonymous or named object by specifying a value for its `int` non-type parameter:

```
int main()
{
    IntType<1> it;
    cout << "IntType<1> objects have value: " << it.value << "\n" <<
            "IntType<2> objects are of a different type "
                    "and have values " << IntType<2>().value << endl;
}
```

22.2.2 Selecting alternatives using templates

Being able to make choices is an essential feature of programming languages. If we want to be able to 'program the compiler' this feature must be present in templates as well. Note again that being able to make choices in templates has *nothing* to do with run-time execution of programs. The essence of template meta programming is that we're *not* using or relying on any executable code in our template meta program. Of course, the result will usually be executable code, but the particular code that is produced must be a function of decisions the compiler can make by itself.

Since template (member) functions are only instantiated when they are actually used, we can even define specializations of functions which are mutually exclusive. I.e., it is possible to define a specialization which may be compilable in one situation, but not in another, and a second specialization which is compilable in the other situation, but not in the first situation. This way code can be tailored to the demands of a concrete situation.

A feature like this cannot be implemented in code. For example, when designing a generic storage class the software engineer may have the intention to store value class type objects as well as objects of polymorphic class types in the storage class. From this point of departure the engineer may conclude that the storage class should contain pointers to objects, rather than objects themselves, and the following code may be conceived of:

```
template <typename Type>
void Storage::add(Type const &obj)
{
    d_data.push_back(
        d_ispolymorphic ?
            obj.clone()
        :
            new Type(obj)
    );
}
```

The intent is to use the `clone()` member function of the `Type` class if `Type` is a polymorphic class and the standard copy constructor if `Type` is a value class.

Unfortunately, this scheme will normally fail to compile as value classes do not define `clone()` member functions and polymorphic base classes should define their copy constructor in the class's private

section. It doesn't matter to the compiler that `clone()` is never called for value classes and the copy constructor is never called for value type classes: it has some code to compile, and can't do that because of missing members. It's as simple as that.

Template meta programming comes to the rescue. Knowing that template functions are only instantiated when used, we design *specializations* of our `add()` function, and provide our class `Storage` with an additional (in addition to `Type` itself) template non-type parameter indicating whether we'll use `Storage` for polymorphic or non-polymorphic classes:

```
template <typename Type, bool isPolymorphic>
class Storage
    ...
```

and we simply define overloaded versions of our `add()` member: one implementing the polymorphic class variant expecting `true` as its argument, and one implementing the value class variant accepting `false` as its argument.

Again we run into a problem: overloading members cannot be based on argument values, only on types. Fortunately there is a way out: in section 22.2.1.1 it was discussed how to convert integral values to types, and knowledge of how to do this now comes in handy. The strategy is to define two overloaded versions: one defining an `IntType<true>` parameter, implementing the polymorphic class and one defining an `IntType<false>` parameter, implementing the polymorphic class. In addition to these overloaded versions of the member function `add()` the member `add()` itself calls the appropriate overloaded member by providing an `IntType` argument, constructed from `Storage`'s template non-type parameter. Here are the implementations:

Declared in `Storage`'s private section:

```
template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<true>)
{
    d_data.push_back(obj.clone());
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<false>)
{
    d_data.push_back(new Type(obj));
}
```

Declared in `Storage`'s public section:

```
template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj)
{
    add(obj, IntType<isPolymorphic>());
}
```

In the above example making a selection was made possible by converting a primitive value to a type and then (since each concrete primitive value may be used to construct a different type) using these types to define overloaded versions of template member functions one of which is then called from a (public) member using `IntType` to construct the appropriate selector type.

Since template members are only instantiated when used, only one of the overloaded private `add()` members is instantiated. Since the other one is never called compilation errors are prevented.

Some software engineers may have second thoughts when thinking about the `Storage` class using pointers to store copies of value classes. Their argument could be that value class objects can be stored by value, rather than by pointer. In those cases we'd like to define the actual type used for storing the

values as either value types or pointer types. Situations like these frequently occur in template meta programming and the following `struct IfElse` may be used to obtain one of two types, depending on a `bool` selector value. First define the *general form* of the template:

```
template<bool selector, typename FirstType, typename SecondType>
struct IfElse
{
    typedef FirstType TypeToUse;
};
```

Then define a specialization for the case where the selector value is `false`. Note that the specialized `struct` has three arguments, matching the template parameters of the above general form. The `template<...>` specification used with specializations merely define what remaining template parameters are present in the specialization:

```
template<typename FirstType, typename SecondType>
struct IfElse<false, FirstType, SecondType>
{
    typedef SecondType TypeToUse;
};
```

The `IfElse` struct uses in its second definition a partial specialization to select the `FalseType` if the `selector` is `false`. In all other cases (i.e., `selector == true`) the less specific generic case is instantiated by the compiler, defining `FirstType` as the `TypeToUse`.

The `IfElse` struct allows us to *templatize structural types*: our `Storage` class may use *pointers* to store copies of polymorphic class type objects, but *values* to store value class type objects.

```
template <typename Type, bool isPolymorphic>
class Storage
{
    typedef typename IfElse<isPolymorphic, Type *, Type>::TypeToUse
            DataType;

    std::vector<DataType> d_data;

    private:
        void add(Type const &obj, IntType<true>);
        void add(Type const &obj, IntType<false>);
    public:
        void add(Type const &obj);
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<true>)
{
    d_data.push_back(obj.clone());
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<false>)
{
    d_data.push_back(obj);
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj)
{
```

```
        add(obj, IntType<isPolymorphic>());
    }
```

The above example uses `IfElse`'s `TypeToUse`, which is a type defined by `IfElse` as either `FirstType` or `SecondType` to define the actual data type to be used for `Storage`'s `std::vector` data type. To prevent long data type definitions `Storage` defines its own type `DataType`.

The remarkable result in this example is that the *structure* of the `Storage` class's data is now depending on its template parameters. Since the `isPolymorphic == true` situation uses different data types than t(isPolymorphic == false) situation, the overloaded private `add()` members can utilize this difference immediately. E.g., `add(Type const &obj, IntType<false>)` now uses direct copy construction to stpre a copy of `obj` in `d_vector`.

It is also possible to select a type from more than two alternatives. In that case, `IfElse` structs can be nested. Remember that these structs never have any effect on the run-time program, which simply is confronted with the appropriate type, conditional to the type that's associated with the selector value. The following example, defining `MapType` as a map having plain types or pointers for either its key or its value type, illustrates this approach:

```
    template <typename Key, typename Value, int selector>
    class Storage
    {
        typedef typename IfElse<
                    selector == 1,              // if selector == 1:
                    map<Key, Value>,            // use map<Key, Value>

                    typename IfElse<
                        selector == 2,          // if selector == 2:
                        map<Key, Value *>,      // use map<Key, Value *>

                        typename IfElse<
                            selector == 3,      // if selector == 3:
                            map<Key *, Value>,  // use map<Key *, Value>
                                                // otherwise:
                            map<Key *, Value *> // use map<Key *, Value *>

                        >::TypeToUse
                    >::TypeToUse
                >::TypeToUse
                MapType;

        MapType d_map;

        public:
            void add(Key const &key, Value const &value);
        private:
            void add(Key const &key, Value const &value, IntType<1>);
            ...
    };
    template <typename Key, typename Value, int selector>
    inline void Storage<selector, Key, Value>::add(Key const &key,
                                                   Value const &value)
    {
        add(key, value, IntType<selector>());
    }
```

The principle used in the above examples is: if different data types are to be used in class templates, depending on template non-type parameters, an `IfElse` struct can be used to define the appropriate type, and overloaded member functions may utilize knowledge about the appropriate types to optimize

their implementations.

Note that the overloaded functions have identical parameter lists as the matching public wrapper function, but add to this parameterlist a specific `IntType` type, allowing the compiler to select the appropriate overloaded version, based on the template's non-type selector parameter.

22.2.3 Templates: Iterations by Recursion

Since there are no variables in template meta programming, there is no way to implement iteration using templates. However, iterations can always be rewritten as recursions, and since recursions *are* supported by templates iterations can always be rewritten as (tail) recursions.

The principle to follow here is:

- Define a specialization implementing the end-condition;
- Define all other steps using recursion.
- Store intermediate values as `enum` values.

Since the compiler will select a more specialized implementation over a more generic one, by the time it reaches the final recursion it will stop the recursion since the specialization will not rely on recursion anymore.

Most readers will be familiar with the recursive implementation of the mathematical '*factorial*' operator, indicated by the exclamation mark (`!`). Factorial `n` (so: `n!`) returns the successive products `n * (n - 1) * (n - 2) * ... * 1`, representing the number of ways `n` objects can be permuted. Interestingly, the factorial operator is usually defined by a *recursive* definition:

```
n! = (n == 0) ?
        1
    :
        n * (n - 1)!
```

To compute `n!` from a template, a template `Factorial` can be defined using a `int n` template non-type parameter, and defining a specialization for the case `n == 0`. The generic implementation uses recursion according to the factorial definition. Furthermore, the `Factorial` template defines an `enum` value '`value`' to contain the its factorial value. Here is the generic definition:

```
template <int n>
struct Factorial
{
    enum { value = n * Factorial<n - 1>::value };
};
```

Note how the expression assigning a value to '`value`' uses constant, compiler determinable values: `n` is provided, and `Factorial<n - 1>()` is computed by *template meta programming*, also resulting in a compiler determinable value. Also note the interpretation of `Factorial<n - 1>::value`: it is the `value` defined by the type `Factorial<n - 1>`; it's *not*, e.g., the value returned by an *object* of that type. There are no objects here, simply values defined by types.

To end the resrsion a specialization is required, which will be preferred by the compiler over the generic implementation when its template arguments are present. The specialization can be provided for the value 0:

```
template <>
struct Factorial<0>
```

```
    {
        enum { value = 1 };
    };
```

The `Factorial` template can be used to determine, compile time, the number of permutations of a fixed number of objects. E.g.,

```
    int main()
    {
        cout << "The number of permutations of 5 objects = " <<
                Factorial<5>::value << "\n";
    }
```

Once again, `Factorial<5>::value` is *not* evaluated run-time, but compile-time. The above statement is therefore run-time equivalent to:

```
    int main()
    {
        cout << "The number of permutations of 5 objects = " <<
                120 << "\n";
    }
```

22.3 Template template parameters

Consider the following situation: a software engineer is asked to design a storage class `Storage` which is able to store data, which may either make and store copies of the data or store the data as received, and which may either use a vector or a linked list as its underlying storage medium. How should the engineer tackle this request?

The engineer's first reaction could be to develop `Storage` as a class having two data members, one being a list, another being a vector, and to provide the constructor with maybe an enum value indicating whether the data itself or new copies should be stored using that enum value to set a series of pointers to member functions to activate the appropriate subset of its private member functions, providing public wrapper functions to hide the use of the pointers to members.

Complex, but doable, until the engineer is confronted with a modification of the original question: now the request states that it should also be possible to use -in the case of new copies- a custom-made allocation scheme, rather than the standard `new` operator, and it should also be possible to use yet another type of container, in addition to the vector and list that were already part of the design. E.g., a `queue` could be preferred or maybe even a `stack`.

It's clear that the approach suggesting to have all functionality provided by the class doesn't scale. The class `Storage` would soon become a monolithic giant which is hard to understand, maintain, test, and deploy.

One of the reasons why the big, all-encompassing class is hard to deploy and understand is that a well-designed class should *enforce constraints*: the design of the class should, by itself, disallow certain operations, violations of which should be detected by the compiler, rather than by a program, crashing or terminating with a fatal error.

Consider the above request. If the class offers both an interface to access the vector data storage *and* an interface to access the list data storage, then it's likely that the class will offer an overloaded `operator[]` member to access elements in the vector. This member, however, will be syntactically present, but semantically invalid when the *list* data storage is selected, which doesn't support `operator[]`.

Sooner or later, *users* of the monolithic all-encompassing class `Storage` will fall into the trap of using `operator[]` even though they've selected the list as the underlying data storage. The compiler won't

be able to detect the error, which will only appear once the program is running, leaving *users* rather than the *engineer* flabbergasted.

The question remains: how should the engineer proceed, when confronted with the above questions? It's time to introduce *policies*.

22.3.1 Policy classes - I

A *policy* defines (in some contexts: prescribes) a particular kind of behavior. In our context a *policy class* defines a certain part of the class interface, and it may define inner types, member functions and data members.

In the previous section a problem of how to create a class which might use a series of allocation schemes was introduced. These allocation schemes all depend on the actual data type to be used, and so the 'template' reflex should kick in: the allocation schemes should probably be defined as template classes, applying the appropriate allocation procedures to the data type at hand. E.g. (using in-class implementations to save some space), the following three allocation classes could be defined:

- No special allocation takes place, `data` is used 'as is':

```
template <typename Data>
class PlainAlloc
{
    template<typename IData>
    friend std::ostream &operator<<(std::ostream &out,
                                    PlainAlloc<IData> const &alloc);
    Data d_data;

    public:
        PlainAlloc()
        {}
        PlainAlloc(Data data)
        :
            d_data(data)
        {}
        PlainAlloc(PlainAlloc<Data> const &other)
        :
            d_data(other.d_data)
        {}
};
```

- The second allocation scheme uses the standard `new` operator to allocate a new copy of the data:

```
template <typename Data>
class NewAlloc
{
    template<typename IData>
    friend std::ostream &operator<<(std::ostream &out,
                                    NewAlloc<IData> const &alloc);
    Data *d_data;

    public:
        NewAlloc()
        :
            d_data(0)
        {}
        NewAlloc(Data const &data)
        :
```

```
            d_data(new Data(data))
        {}
        NewAlloc(NewAlloc<Data> const &other)
        :
            d_data(new Data(*other.d_data))
        {}
        ~NewAlloc()
        {
            delete d_data;
        }
};
```

- The third allocation scheme uses the placement new operator (see section 8.1.4), requesting memory from a common pool of bytes (the implementation of the member `request()`, obtaining the required amount of memory, is left as an exercise to the reader):

```
template<typename Data>
class PlacementAlloc
{
    template<typename IData>
    friend std::ostream &operator<<(std::ostream &out,
                                    PlacementAlloc<IData> const &alloc);
    Data *d_data;

    static char s_commonPool[];
    static char *s_free;

    public:
        PlacementAlloc()
        :
            d_data(0)
        {}
        PlacementAlloc(Data const &data)
        :
            d_data(new(request()) Data(data))
        {}
        PlacementAlloc(PlacementAlloc<Data> const &other)
        :
            d_data(new(request()) Data(*other.d_data))
        {}
        ~PlacementAlloc()
        {
            d_data->~Data();
        }
    private:
        static char *request();
};
```

The above three classes define *policies* that may be selected by the user of the class `Storage`, introduced in the previous section. In addition to this, additional allocation schemes could be implemented by the user as well.

In order to be able to apply the proper allocation scheme to the class `Storage` it should also be designed as a class template. The class will also need a template type parameter allowing users to specify the data type.

It would be possible to specify the data type with the specification of the allocation scheme, resulting in code like:

```
template <typename Data, typename Scheme>
```

```
class Storage ...
```

and then use `Storage`, e.g., as follows:

```
Storage<string, NewAlloc<string> > storage;
```

However, this implementation is unnecessarily complex, as it requires the user to specify the data type twice. Instead, the allocation scheme should be specified using a third type of template parameter, not requiring the user to specify the data type with the allocation scheme to use. This third type of template parameter (in addition to the well-known *template type parameter* and *template non-type parameter*) is the *template template parameter*.

22.3.2 Policy classes - II: template template parameters

Template template parameters allow us to specify a *class template* as a template parameter. By specifying a class template, it is possible to add a certain kind of behavior, called *policy* to an existing class template.

Consider the class `Storage`, introduced at the beginning of this section, and consider the allocation classes discussed in the previous section. To specify an *allocation policy* the class `Storage` starts its definition as follows:

```
template <typename Data, template <typename> class Policy>
class Storage ...
```

The second template parameter is the *template template parameter*. It contains the following elements:

- The keyword `template` starts the definition of a template template parameter;
- It is followed, between pointed brackets, a list of template parameters that must be specified for the template template parameter. These parameters *may* be given names, but these names cannot be used in the subsequent template definition, and are therefore usually omitted. On the other hand, providing formal names may help the reader of the template to understand the kinds of templates that may be specified as template template parameters.
- Template template parameters must match, in numbers and types (template type parameter, template non-type parameter, template template parameter) the template parameters that must be specified for the policy.
- Following the bracketed list the keyword `class` is provided. In this case, `typename` cannot be used.
- All parameter values may be provided with default values, as shown in the following example of a hypothetical class template:

```
template
<
    template
    <
        typename = std::string,
        int = 12,
        template
        <
            typename = int
        >
        class Inner = std::vector
    >
```

```
        class Policy
    >
    class Demo
    {
        ...
    };
```

Since the policy class should be an inherent part of the class under consideration, it is often deployed as a base class. So, `Policy` becomes a base class of `Storage`. Moreover, the policy should operate on the data type to be used with the class `Storage`. Therefore the policy is handed that data type as well. From this we obtain the following setup:

```
template <typename Data, template <typename> class Policy>
class Storage: public Policy<Data>
```

This scheme allows us to use the policy's members when implementing the members of the class `Storage`.

Now the allocation classes do not really offer many useful members: apart from the extraction operator, no immediate access to the data is offered. This can easily be repaired by providing some additional members. E.g., the class `NewAlloc` could be extended with the following operators, allowing access to and modification of stored data:

```
    operator Data &()   // optionally add a 'const' member too
    {
        return *d_data;
    }
    NewAlloc &operator=(Data const &data)
    {
        *d_data = data;
    }
```

Other allocation classes can be provided with comparable members.

The next step is to use the allocation schemes in some real code. The following example shows how a storage can be constructed for a data type to be specified and an allocation scheme to be specified. First, define a class `Storage`:

```
template <typename Data, template <typename> class Policy>
class Storage: public std::vector<Policy<Data> >
{
};
```

That's all there is. All required functionality is offered by the `vector` base class, while the policy is 'factored into the equation' via the template template parameter. Here's an example of its use:

```
Storage<std::string, NewAlloc> storage;

copy(istream_iterator<std::string>(cin), istream_iterator<std::string>(),
        back_inserter(storage));

cout << "Element index 1 is " << storage[1] << endl;
storage[1] = "hello";

copy(storage.begin(), storage.end(),
     ostream_iterator<NewAlloc<std::string> >(cout, "\n"));
```

Following the construction of a `Storage` object, the STL `copy()` function can be used in combination with the *back_inserter* iterator to add some data to `storage`. Its elements can be both accessed and modified directly using the index operator, and then `NewAlloc<std::string>` objects are inserted into `cout`, again using the STL `copy()` algorithm.

Interestingly, this is not the end of the story. After all, the intention was to create a class allowing us to specify the *storage type* as well. What if we don't want to use a `vector`, but instead would like to use a `list`?

It's easy to change `Storage`'s setup so that a completely different storage type can be used on request, say a `list` or a `deque`. To implement this, the storage class is parameterized as well, again using a template template parameter, that could be given a default value too, as shown in the following redefinition of `Storage`:

```
template <typename Data, template <typename> class Policy,
                         template <typename> class Container =
                                                   std::vector>
class Storage: public Container< Policy<Data> >
{
};
```

The earlier example in which a `Storage` object was used can be used again, without any modifications, for the above redefinition. It clearly can't be used with a `list` container, as the `list` lacks `operator[]`. But that's immediately recognized by the compiler, producing an error if an attempt is made to use `operator[]` on, e.g., a `list`[1].

22.3.2.1 The destructor of Policy classes

In the previous section policy classes are used as base classes of template classes resulting in the interesting observation that a policy class actually serves as a *base class* of a derived class Since a policy class may act as a base class, it is thinkable that a pointer or reference to a policy class is used to point or refer to the derived class using the policy.

This situation, although legal, should be avoided for various reasons:

- Destruction of a derived class object using the base class's destructor requires the implementation of a virtual destructor;
- A virtual destructor introduces overhead to a class that normally has no data members, but merely defines behavior: suddenly a `vtable` is required as well as a data member: a pointer to the vtable;
- Virtual member functions somewhat reduce the efficiency of code; thus virtual member functions using *dynamic polymorphism*, somewhat counteract the *static polymorphism* offered by templates;
- Virtual member functions in templates may result in *code bloat*: once an instantiation of a class's member is required, the class's vtable and *all* its virtual members must be implemented too.

To avoid these drawbacks, it is good practice to prevent using references or pointers to policy classes to refer or point to derived class objects. This is accomplished by providing policy classes with *nonvirtual protected destructors*. Since the destructor is non-virtual there is no implementation penalty in reduced efficiency or memory overhead, and since it is protected users cannot refer to classes derived from the policy class using a pointer or reference to the policy class.

[1] A complete example showing the definition of the allocation classes and the class `Storage` as well as its use is provided in the Annotation's distribution in the file `yo/advancedtemplates/examples/storage.cc`.

22.3.3 Structure by Policy

Policy classes usually define behavior, not structure. I.e., policy classes are used to parameterize some aspect of the behavior of classes that are derived from them. However, different policies may imply the use of different data members. Thus a policy class may be used to define both behavior and structure.

By providing a well-defined interface a class derived from a policy class may define member specializations using the different structures of policy classes to their advantage. For example, a plain pointer-based policy class could offer its functionality by resorting to **C**-style pointer juggling, whereas a `vector`-based policy class could use the vector's members directly.

In this situation a generic class template `Size` could be designed expecting a container-like policy using features commonly found in containers, defining the data (and hence the structore) of the container specified in the policy. E.g.:

```
template <typename Data, template <typename> class Container>
struct Size: public Container<Data>
{
    size_t size()
    {                           // relies on the container's 'size()'
                                // note: can't use 'this->size()'
        return Container<Data>::size();
    }
};
```

Next, a specialization can be defined to accomodate the specifics of a much simpler storage class using, e.g., plain pointers (the implementation capitalizes on `first` and `second`, data members of `std::pair`. Cf. the example at the end of this section):

```
template <typename Data>
struct Size<Data, Plain>: public Plain<Data>
{
    size_t size()
    {                           // relies on pointer data members
        return this->second - this->first;
    }
};
```

Depending on the intentions of the template's author other members could be implemented as well.

To use the above templates for real, a generic wrapper class can now be constructed: depending on the actual storage type that is used (e.g., a `std::vector` or some plain storage class) it will use the matching `Size` template to define its structure:

```
template <typename Data, template <typename> class Store>
class Wrapper: public Size<Data, Store>
{};
```

The above classes could now be used as follows (*en passant* showing an extremely basic `Plain` class):

```
#include <iostream>
#include <vector>

template <typename Data>
struct Plain: public std::pair<Data *, Data *>
{};

int main()
```

```
    {
        Wrapper<int, std::vector> wiv;
        std::cout << wiv.size() << "\n";

        Wrapper<int, Plain> wis;
        std::cout << wis.size() << "\n";
    }
```

The `wiv` object now defines vector-data, the `wis` object merely defines a `std::pair` object's data members.

22.4 Trait classes

Scattered over the `std` namespace *trait classes* are found. E.g., most **C++** programmers will have seen the compiler mentioning '`std::char_traits<char>`' when performing an illegal operation on `std::string` objects, as in `std::string s(1)`.

Trait classes are used to make compile-time decisions about types. Traits classes allow the software engineer to apply the proper code to the proper data type, be it a pointer, a reference, or a plain value, all maybe in combination with `const`. Moreover, the specification of the particular type of data to be used does not have to be made by the template writer, but can be inferred from the actual type that is specified (or implied) when the template is used.

Trait classes allow the software engineer to develop a `template <typename Type1, typename Type2, ...>` without the need to specify many specializations covering all combinations of, e.g., values, (const) pointers, or (const) references, which would soon result in an unmaintainable exponential explosion of template specializations (e.g., allowing these five different actual types for each template parameter already results in 25 combinations when two template type parameters are used: each must be covered by potentially different specializations).

Having available a trait class, the actual type can be inferred compile time, allowing the compiler to deduct whether or not the actual type is a pointer, a pointer to a member, a const pointer, and make comparable deductions in case the actual type is, e.g., a reference type. This in turn allows us to write templates that define types like `argument_type`, `first_argument_type`, `second_argument_type` and `result_type`, which are required by several generic algorithms (e.g., count_if()).

A trait class usually performs no behavior. I.e., it has no constructor and no members that can be called. Instead, it defines a series of types and `enum` values that have certain values depending on the actual type that is passed to the trait class template. The compiler uses one of a set of available specializations to select the one appropriate for an actual template type parameter.

The generic point of departure when defining a trait template is a plain vanilla `struct`, defining the characteristics of a plain value type, e.g., an `int`. This sets the stage for specific specializations, modifying the characteristics for any other type that could be specified for the template.

To make matters concrete, assume the intent is to create a trait class `BasicTraits` telling us whether a type is a plain value type, a pointer type, or a reference type (all of which may or may not be `const` types).

Moreover, whatever the actual type that's provided, we want to be able to determine the 'plain' type (i.e., the type without any modifiers, pointers or references), the 'pointer type' and the 'reference type', allowing us to define in all cases, e.g., a reference to its built-in type, even though we passed a const pointer to that type.

Our point of departure, as mentioned, is a plain `struct` defining the required parameter. E.g., something like:

```
        template <typename T>
        struct Basic
```

```
    {
        typedef T Type;
        enum
        {
            isValue = true,
            isPointer = false,
            isConst = false
        };
    };
```

However, often decisions about types can be made using constant logical expressions. Note that the above definition does not contain a '`isReference`' enumeration value. Such a value is not required as it is implied by the expression `not isPointer and not isValue`.

Although some conclusions can be drawn by combining various `enum` values, it is good practice to provide a full implementation of trait classes, not requiring its users to construct these logical expressions themselves. Therefore, the basic decisions in a trait class are usually made by a nested trait class, leaving the task of creating appropriate logical expressions to a surrounding trait class.

So, the `struct Basic` defines the generic form of our inner trait class. Specializations handle specific details. E.g., a pointer type is recognized by the following specialization:

```
    template <typename T>
    struct Basic<T *>
    {
        typedef T Type;
        enum
        {
            isValue = false,
            isPointer = true,
            isConst = false
        };
    };
```

whereas a pointer to a const type is matched with the next specialization:

```
    template <typename T>
    struct Basic<T const *>
    {
        typedef T Type;
        enum
        {
            isValue = false,
            isPointer = true,
            isConst = true
        };
    };
```

Several other specializations should be defined: e.g., recognizing `const` value types or reference types. Eventually all these specializations wind up being nested `structs` of an outer class `BasicTraits`, offering the public traits class interface. The outline of the outer trait class is:

```
template <typename TypeParam>
class BasicTraits
{
    // Define specializations of the template 'Base' here

    public:
```

```
        typedef typename Basic<TypeParam>::Type ValueType;
        typedef ValueType *PtrType;
        typedef ValueType &RefType;

        enum
        {
            isValueType = Basic<TypeParam>::isValue,
            isPointerType = Basic<TypeParam>::isPointer,
            isReferenceType = not Basic<TypeParam>::isPointer and
                              not Basic<TypeParam>::isValue,
            isConst = Basic<TypeParam>::isConst
        };
};
```

A trait class template can be used to obtain the proper type, irrespective of the template type argument provided, or it can be used to select the proper specialization, depending on, e.g., the `const`-ness of a template type. The following statements serve as an illustration:

```
cout << BasicTraits<int>::isPointerType << " " <<
        BasicTraits<int *>::isPointerType << " " <<
        BasicTraits<int>::isConst << " " <<
        BasicTraits<int>::isReferenceType << " " <<
        BasicTraits<int &>::isReferenceType << " " <<
        BasicTraits<int const>::isConst << " " <<
        BasicTraits<int const *>::isPointerType << " " <<
        BasicTraits<int const *>::isConst << " " <<
        endl;

BasicTraits<int *>::ValueType value = 12;
int *otherValue = &value;
cout << *otherValue << endl;
```

22.4.1 Distinguishing class from non-class types

In the previous section the `TypeTrait` trait class was developed. Using specialized versions of a nested `struct Type` modifiers, pointers, references and values could be distinguished.

Knowing whether a type is a class type or not (e.g., the type represents a primitive type) could also be a useful bit of knowledge to a template developer. E.g, the class template developer might define a specialization for a member knowing the template's type parameter is a class type (maybe using some member function that should be available) and another specialization for non-class types.

This section addresses the question how a trait class can distinguish class types from non-class types.

In order to distinguish classes from non-class types a distinguishing feature that can be used compile-time must be found. It may take some thinking to find such a distinguishing characteristic, but a good candidate eventually is found in the pointer to members syntactic construct, which is available only for classes. Using the pointer to member construct as the distinguishing characteristic, we now look for a construction which uses the pointer to member if available, and does something else if the pointer to member construction is not available.

Note again the rule of thumb that works so well for template meta programming: define a generic situation, and then specialize for the situations you're interested in. It's not a trivial task to apply this rule of thumb here: how can we distinguish a pointer to a member from 'a generic situation', not being a pointer to a member? Fortunately, such a distinction is possible: a function template can be provided with a parameter which is a pointer to a member function (defining the 'specialization' case), and another function template can be defined so that it accepts any argument. The compiler will then *select* the latter function in all situations but those in which the provided type is actually a class type, and thus a type which *may* support a pointer to a member.

Note that the compiler will not *call* the functions: we're talking compile-time here, and all the compiler does is to *select* the appropriate function, in order to be able to evaluate a constant expression (defining the value of, e.g, the enum value `isClass`).

So, our function template will be something like:

```
template <typename ClassType>
static (returntype)   fun(void (ClassType::*)());
```

Note that in this function '`(returntype)`' is not yet specified. It will be shortly.

The question about what the return type should be will be answered shortly. Arbitrarily the function's parameter defines a pointer to a member returning `void`. Note that there's *no* need for such a function to exist for the concrete class-type that's specified with the traits class, since all the compiler will do is *select* this function if a class-type was provided to the trait class in which `fun()` will be nested. In line with this: `fun()` is only declared, not defined. Furthermore note that `fun()` is declared as a *static* member of the trait class, so that there's no need for an actual object when `fun()` is called.

So far for the class-types. What about the non-class types? For those types a (generic) alternative must be found, one the compiler will select when the actual type is not a class type. Again, the language offers a 'worst case' solution in the *ellipsis* parameter list. The ellipsis is a final resort the compiler may turn to if everything else fails. It's not only used to define (the in **C++** deprecated) functions having a variable number of arguments, but it's also used to define the catch-all exception `catch` clause. Therefore, the 'generic case' can be defined as follows:

```
template <typename NonClassType>
static (returntype)   fun(...);
```

Note that it would be an error to define the generic alternative as a function expecting an `int`. The compiler, when confronted with alternatives, will favor the simplest, most specified alternative over a more complex, generic one. So, when providing `fun()` with an argument it will select `int` when possible, given the nature of the used argument.

The question now becomes: what argument can be used for both a pointer to a member and the generic situation? Actually, there is such a 'one size fits all' argument: 0. The value 0 can be used as argument value to initialize not only primitive types, but also to initialize pointers and pointers to members. Therefore, `fun` will be called as `fun<Type>(0)`, with `Type` being the template type parameter of the trait class. Here, `Type` must be specified since the compiler will not be able to determine `fun`'s template type parameter when `fun(...)` is selected.

Now for the return type: the return type cannot be a simple value (like `true` or `false`). When using a simple value the `isClass` enum value cannot be defined, since

```
enum { isClass = fun<Type>(0) } ;
```

needs to be evaluated to obtain `fun`'s return value, which is clearly not possible as enum values *must* be determined compile-time.

To allow a compile-time definition of `isClass`'s value the solution must be sought in an expression that discriminates between `fun<Type>(...)` and `fun<Type>(void (Type::*)())`. In situations like these `sizeof` is our tool of choice. The `sizeof` operator is evaluated compile-time, and so by defining return types that differ in their sizes it is possible to discriminate compile-time among the two `fun()` alternatives.

The `char` type is by definition a type having size 1. By defining another type containing two consecutive `char` values a bigger type is obtained. Now `char [2]` is not a type, but `char[2]` can be defined as a data member of a `struct` which will thus have a size exceeding 1. E.g.,

```
struct Char2
```

```
{
    char data[2];
};
```

`Char2` can be defined as a nested type within our traits class, and the two `fun` declarations become:

```
template <typename ClassType>
static Char2 fun(void (ClassType::*)());

template <typename NonClassType>
static char fun(...);
```

This, in turn enables us to specify an expression that can be evaluated compile time, allowing the compiler to determine `isClass`'s value:

```
enum { isClass = sizeof(fun<Type>(0)) == sizeof(Char2) };
```

Note, however, that no `fun()` function template *ever* makes it to the instantiation stage, but the compiler nonetheless is able to infer what `fun`'s return type will be, given a concrete template type argument. This inference is then used by the compiler to determine the truth of an expression, in turn enabling the compiler to compute the required compile-time constant value `isClass`, allowing us to determine whether a certain type is or is not a class type. Marvelous!

22.4.2 Available type traits (C++0x)

The C++0x standard offers the header file `type_traits` allowing template meta programs to identify and modifiy characteristics of types.

All facilities offered in `type_traits` are defined in the `std` namespace (omitted from the examples given below) allowing programmers to

- determine whether a type is an lvalue reference (`is_lvalue_reference<typename Type>::value`);
- determine whether a type is an rvalue reference (`is_rvalue_reference<typename Type>::value`);
- determine whether a type is a reference (`is_reference<typename Type>::value`);
- determine whether a type is a signed type (`is_signed<typename Type>::value`);
- determine whether a type is an unsigned type (`is_unsigned<typename Type>::value`);
- determine whether a type is *plain old data* (e.g., a struct not having members) (`is_pod<typename Type>::value`);
- determine whether a type has a trivial default constructor (`has_trivial_default_constructor<typename Type>::value`); This concept of a trivial member is also used in the next few type trait facilities.
- determine whether a type has a trivial copy constructor (`has_trivial_copy_constructor<typename Type>::value`);
- determine whether a type has a trivial destructor (`has_trivial_destructor<typename Type>::value`);

- determine whether a type has a trivial assignment operator
  (`has_trivial_assign<typename Type>::value`);
- determine whether a type has a constructor not throwing exceptions
  (`has_nothrow_default_constructor<typename Type>::value`);
- determine whether a type has a destructor not throwing exceptions
  (`has_nothrow_destructor<typename Type>::value`);
- determine whether a type has a copy constructor not throwing exceptions
  (`has_nothrow_copy_constructor<typename Type>::value`);
- determine whether a type has an assignment operator operator not throwing exceptions
  (`has_nothrow_assign<typename Type>::value`);
- determine whether a type `Base` is a base class of another type `Derived`
  (`is_base_of<typename Base, typename Derived>::value`);
- determine whether a type `From` may be converted to a type `To` (e.g., using a `static_cast`)
  (`is_convertible<typename From, typename To>::value`);
- conditionally define `Type` if `cond` is true
  (`enable_if<bool cond, typename Type>::type`);
- conditionally use `TrueType` if `cond` is true, `FalseType` if not
  (`conditional<bool cond, typename TrueType, typename FalseType>::type`);
- remove a reference from a type
  (`remove_reference<typename Type>::type`);
- add an lvalue reference to a type
  (`add_lvalue_reference<typename Type>::type`);
- add an rvalue reference to a type
  (`add_rvalue_reference<typename Type>::type`);
- construct an unsigned type
  (`make_unsigned<typename Type>::type`);
- construct a signed type
  (`make_signed<typename Type>::type`);

22.5 More conversions to class types

22.5.1 Types to types

Although *class* templates may be partially specialized, *function* templates may not. At times that can be annoying. Assume a function template is available implementing a certain unary operator that could be used with the `transform` (cf. section 19.1.63) generic algorithm:

```
template <typename Return, typename Argument>
Return chop(Argument const &arg)
{
    return Return(arg);
}
```

Furthermore assume that if `Return` is `std::string` then the specified implementation should not be used. Rather, with `std::string` a second argument `1` should always be provided (e.g., if `Argument` is a **C++** string, a `std::string` is returned holding a copy of the function's argument, except for the argument's first character, which is chopped off).

Since `chop()` is a function, it is not possible to use a partial specialization. So it is not possible to specialize for `std::string` as attempted in the following erroneous implementation:

```
template <typename Argument>
std::string chop<std::string, Argument>(Argument const &arg)
{
    return string(arg, 1);
}
```

it *is* possible to use overloading, though. Instead of using partial specializations *overloaded function templates* could be designed:

```
template <typename Return, typename Argument>
Return chop(Argument const &arg, Argument )
{
    return Return(arg);
}

template <typename Argument>
std::string chop(Argument const &arg, std::string )
{
    return string(arg, 1);
}
```

This way it *is* possible to distinguish the two cases, but at the expense of a more complex function call (e.g., maybe requiring the use of the `bind2nd()` binder (cf. section 18.1.4) to bind the second argument to a fixed value) as well as the need to provide a (possibly expensive to construct) dummy argument to allow the compiler to choose among the two overloaded function templates.

Alternatively, overloaded versions *could* use the `IntType` template (cf. section 22.2.1.1) to select the proper overloaded version. E.g., `IntType<0>` could be defined as the type of the second argument of the first overloaded `chop()` function, and `IntType<1>` could be used for the second overloaded function. From the point of view of program efficiency this is an attractive option, as the provided `IntType` objects are extremely lightweight: they contain no data at all. But there's also an obvious disadvantage: there is no intuitively clear association between on the one hand the `int` value used and on the other hand the intended type.

In situations like these it is more attractive to use another lightweight solution. Instead of using an arbitrary int-to-type association, an intuitively clear and automatic type-to-type association is used. The `struct TypeType` is a lightweight type wrapper, much like `IntType` is a lightweight wrapper around an `int`. Here is its definition:

```
template <typename T>
struct TypeType
{
    typedef T Type;
};
```

This too is a lightweight type as it doesn't have any data fields either. `TypeType` allows us to use a natural type association for `chop()`'s second argument. E.g, the overloaded functions can now be defined as follows:

```
template <typename Return, typename Argument>
Return chop(Argument const &arg, TypeType<Argument> )
{
    return Return(arg);
}
```

```
template <typename Argument>
std::string chop(Argument const &arg, TypeType<std::string> )
{
    return std::string(arg, 1);
}
```

Using the above implementations any type can be specified for `Result`. If it happens to be a `std::string` the correct overloaded version is automatically selected. E.g.,

```
template <typename Result>
Result chopper(char const *txt)
{
    return chop(std::string(txt), TypeType<Result>());
}
```

Using `chopper()`, the following statement will produce the text '`ello world`':

```
cout << chopper<string>("hello world") << endl;
```

22.5.2 An empty type

At times (cf. section 22.6) an empty `struct` is a useful little tool. It can be used as a *type* acting analogously to the `ASCII-Z` (final 0-byte) in **C**-strings or as a 0-pointer to indicate the end of a linked list. Its definition is simply:

```
struct NullType
{};
```

22.5.3 Type convertability

In what situations can a type `T` be used as a 'stand in' for another type `U`? Since **C++** is a strongly typed language the answer is surprisingly simple: `T`s can be used instead of `U`s if a `T` is accepted as argument in cases where `U`s are requested.

This reasoning is behind the following class which can be used to determine whether a type `T` can be used where a type `U` is expected. The interesting part, however, is that no code is actually generated or executed: all decisions can be made by the compiler.

In the second part of this section it will be shown how the code developed in the first part can be used to detect whether a class `B` is a base class of another clas `D`. The code developed here closely follows the example provided by Alexandrescu (2001, p. 35).

First, a function is conceived of that will accept a type `U` to which an alternative type `T` will be compared. This function returns a value of the as yet unsepcified type `Convertible`:

```
Convertible test(U const &);
```

The function `test()` is not implemented; it is merely declared. The idea is that if a type `T` can be used instead of a type `U` it can be passed as argument to the above `test()` function.

If `T` cannot be used where a `U` is expected, then the above function will not be used by the compiler. Of course, getting a compiler error is not the kind of 'answer' we're looking for and so the next question is what alternative we can offer to the compiler in cases where a `T` cannot be used as argument to the above function.

C (and **C++**) offer a very general parameter list, a parameter list that will always be considered acceptable. This parameter list consists of the *ellipsis* which actually is the *worst* situation the compiler may encounter: if everything else fails, then the function defining an ellipsis as its parameter list is selected. Normally that's not a productive and type-safe alternative, but in the current situation it is *exactly* what is needed. When confronted with two alternative function calls, one of which defines the ellipsis parameter list, the compiler will only select the function defining the ellipsis if the alternative(s) can't be used. So we declare an alternative function `test()`, having the ellipsis as its parameter list, and returning another type, e.g., `NotConvertible`:

```
NotConvertible test(...);
```

Now, if code passes a value of type `T` to the `test` function, the return type will be `Convertible` if `T` can be converted to `U`, and `NotConvertible` if conversion is not possible.

Two problems still need to be solved: how do we obtain a `T` argument? The problems here are firstly, that it might not be possible to define a `T`, as a type `T` might hide all its constructors and secondly, how can the two return values be distinguished?

Although it might be impossoble to construct a `T` object, it is fortunately not necessary to construct a `T`. After all, the intent is to decide *compile time* whether a type is convertible to another type and not actually to construct such a `T` value. So another function is *declared*:

```
T makeT();
```

This mysterious function has the magical power of enticing the compiler into thinking that a `T` object will come out of it. So, what will happen when the compiler is shown the following code:

```
test(makeT())
```

If `T` can be converted to `U` the first function `Convertible test(U const &)` will be selected by the compiler; otherwise the function `NotConvertible test(...)` will be selected.

If it is now possible to distinguish `Convertible` from `NotConvertible` compile-time then it is possible to determine whether `T` is convertible to `U`.

Since `Convertible` and `NotConvertible` are values, their sizes are known. If these sizes differ, then the `sizeof` operator can be used to distinguish the two types; hence it is possible to determine which `test()` function was selected and hence it is known whether `T` can be converted to `U` or not. E.g., if the following expression evaluates as `true` `T` is convertible to `U`:

```
sizeof(test(makeT())) == sizeof(Convertible);
```

The size of a `char` is well known. By definition it is 1. Using a `typedef` `Convertible` can be defined as a synonym of `char`, thus having size 1. Now `NotConvertible` must be defined so that it has a different type. E.g.,

```
struct NotConvertible
{
    char array[2];
};
```

Note there that a simple `typedef char NotConvertible[2]` does not work: functions cannot return arrays, but they can return arrays embedded in a structs.

The above can be wrapped up in a template class, having two template type parameters:

```
template <typename T, typename U>
```

```
class Conversion
{
    typedef char Convertible;
    struct NotConvertible
    {
        char array[2];
    };

    static T makeT();
    static Convertible test(U const &);
    static NotConvertible test(...);

    public:
        enum { exists = sizeof(test(makeT())) == sizeof(Convertible) };
        enum { sameType = 0 };
};

template <typename T>
class Conversion<T, T>
{
    public:
        enum { exists = 1 };
        enum { sameType = 1 };
};
```

The above class *never* results in *any run-time* execution of code. When used, it merely defines the values 1 or 0 for its `exist` enum value, depending whether the conversion exists or not. The following example writes `1 0 1 0` when run from a `main()` function:

```
cout <<
    Conversion<ofstream, ostream>::exists << " " <<
    Conversion<ostream, ofstream>::exists << " " <<
    Conversion<int, double>::exists << " " <<
    Conversion<int, string>::exists << " " <<
    endl;
```

22.5.3.1 Determining inheritance

Now that `Conversion` has be defined it's easy to determine whether a type `Base` is a (public) base class of a type `Derived`. To determine inheritance convertability of (const) pointers is examined. `Derived const *` can be converted to `Base const *` if:

- Both types are identical;
- `Base` is a public and unambiguous base class of `Derived`;
- and also, but usually not intended: if `Base` is void.

Preventing the latter, inheritance is determined by inspecting `Conversion<Derived const *, Base const *>::exists`:

```
#define BASE_1st_DERIVED_2nd(Base, Derived) \
    (Conversion<Derived const *, Base const *>::exists && \
     not Conversion<Base const *, void const *>::sameType)
```

If code should not consider a class to be its own base class, then the following stricted test is possible, which adds a test for type-equality:

```
#define BASE_1st_DERIVED_2nd_STRICT(Base, Derived) \
    (BASE_1st_DERIVED_2nd(Base, Derived) && \
     not Conversion<Base const *, Derived const *>::sameType)
```

The following example writes `1:  0, 2:  1, 3:  0, 4:  1, 5:  0` when run from a `main()` function:

```
cout << "\n" <<
    "1: " << BASE_1st_DERIVED_2nd(ofstream, ostream) << ",  " <<
    "2: " << BASE_1st_DERIVED_2nd(ostream, ofstream) << ",  " <<
    "3: " << BASE_1st_DERIVED_2nd(void, ofstream) << ",  " <<
    "4: " << BASE_1st_DERIVED_2nd(ostream, ostream) << ",  " <<
    "5: " << BASE_1st_DERIVED_2nd_STRICT(ostream, ostream) << " " <<
    endl;
```

22.6 Template TypeList processing

This section serves two purposes. On the one hand it illustrates capabilities of the various meta-programming capabilities of templates, which can be used as a source for inspiration when developing your own templates. On the other hand, it culminates in a concrete example, showing some of the power template meta-programming has.

This section itself was inspired by Andrei Alexandrescu's (2001) book **Modern C++ design**, and much of this section's structure borrows from Andrei's coverage of *typelists*.

A typelist is a very simple `struct`: like a `std::pair` it consists of two elements, although in this case the typelist does not contain data members, but type definitions. It is defined as follows:

```
template <typename First,  typename Second>
struct TypeList
{
    typedef First Head;
    typedef Second Tail;
};
```

The typelist allows us to store any number of types using a recursive definition. E.g., the three types `char, short, int` may be stored as follows:

```
TypeList<char, Typelist<short, int> >
```

Although this is a possible representation, usually `NullType` (cf. section 22.5.2) is used as the final type, acting comparably to a 0-pointer. Using `NullType` the above three types are represented as follows:

```
TypeList<char,
    TypeList<short,
        TypeList<int, NullType> > >
```

This way to represent lists of types may be accepted by the compiler, but usually not as easily by programmers, who frequently have a hard time putting in the right number of parentheses. Alexandrescu suggest to ease the burden by defining a series of *macros*, even though macros are generally deprecated in **C++**. The `TYPELIST` macros suggested by Alexandrescu allow us to define typelists for varying numbers of types, and they are easily expanded if accommodating larger numbers of types is necessary. Here are the definitions of the first five `TYPELIST` macros:

```
#define TYPELIST_1(T1)              TypeList<T1, NullType >
```

```
#define TYPELIST_2(T1, T2)          TypeList<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3)      TypeList<T1, TYPELIST_2(T2, T3) >
#define TYPELIST_4(T1, T2, T3, T4)  TypeList<T1, TYPELIST_3(T2, T3, T4) >
#define TYPELIST_5(T1, T2, T3, T4, T5)  TypeList<T1, TYPELIST_4(T2, T3, T4, T5) >
```

Note the recursive nature of these macro definitions: recursion is how template meta programs perform iteration and in the upcoming sections implementations heavily depend on recursion. With all solutions to the problems covered by the coming sections a verbal discussion is provided explaining the philosophies that underlie the recursive implementations.

Of course, even here macros are ugly. The macro processor will be confused if a type is somewhat complex, like `Wrap<HEAD, idx>`. Fortunately situations like these can be prevented using a simple `typedef`. E.g., `typenef Wrap<HEAD, idx> HEADWRAP` and then using `HEADWRAP` instead of the full type definition.

22.6.1 The length of a TypeList

To obtain the length of a typelist the following algorithm can be used:

- If the typelist is empty, its size is zero
- If the typelist is non-empty, its size equals 1 plus the size of its tail.

Note how recursion is used to define the length of a typelist. In 'normal' **C++** code this recursion could be implemented as well, e.g., to determine the length of a plain **C** (ascii-Z) string, resulting in something like:

```
size_t c_length(char const *cp)
{
    return *cp == 0 ? 0 : 1 + c_length(cp + 1);
}
```

In the context of template meta programming the alternatives that are used to execute or terminate recursion are never written in one implementation, but instead *specializations* are used: each specialization implements an alternative.

The length of a typelist will be determined by a `struct ListSize` ordinarily expecting a typelist as its template type parameter. It's merely declared, since it turns out that only its specializations are required:

```
template <typename TypeList>
struct ListSize;
```

Following the above algorithm *specializations* are now constructed:

- If the `ListSize`'s type is empty (i.e., a `NullType`), its size is 0:

```
template <>
struct ListSize<NullType>
{
    enum {  size = 0  };
};
```

- Otherwise, its size will be 1 plus the size of the tail of a `TypeList`:

```
template <typename Head, typename Tail>
```

```
struct ListSize<TypeList<Head, Tail> >
{
    enum {  size = 1 + ListSize<Tail>::size  };
};
```

That's all. The size of any typelist can now easily be determined. E.g., assuming all required headers (templates, iostream) have been included then the following statement will (of course) display the value 3:

```
std::cout << ListSize<TYPELIST_3(int, char, bool)>::size << "\n";
```

22.6.2 Searching a TypeList

To determine whether a type (called the *searchtype* below) is present in a given typelist, an algorithm is used that will either define 'index' -1 (if the searchtype is not an element of the typelist) or it will define 'index' as the index of the first occurrence of the searchtype in the typelist. The following algorithm is used:

- If the typelist is equal to `NullType`, define 'index' as -1;
- If the typelist's head element equals the searchtype, define 'index' as 0;
- Otherwise define 'index' as follows:
  - If searching the searchtype in the typelist's tail results in a index -1, then searchtype is not an element of the typelist, and the (current) index will be set to -1 as well;
  - Otherwise, searchtype was found in the typelist's tail, and the current index will be set to 1 + the index obtained for searchtype on the typelist's tail.

The implementation again sets out with the declaration of a struct: `ListSearch` expects two template type parameters: searchtype's type and a typelist:

```
template <typename SearchType, typename TypeList>
struct ListSearch;
```

Next, specializations are defined implementing the above alternatives:

- If the typelist is empty, define 'index' as -1:

```
template <typename SearchType>
struct ListSearch<SearchType, NullType>
{
    enum { index = -1 };
};
```

- If the typelist's head element equals the searchtype, define 'index' as 0. Note how the test is performed by specifying `SearchType` twice:

```
template <typename SearchType, typename Tail>
struct ListSearch<SearchType, TypeList<SearchType, Tail> >
{
    enum { index = 0 };
};
```

- Otherwise define 'index' as either -1 (searchtype wasn't found in the typelist's tail) or as 1 + the index obtained from searching the typelist's tail. Note that the implementation uses a *private enum value* `tmp` to store the index value obtained from searching the typelist's tail for searchtype:

```
template <typename SearchType, typename Head, typename Tail>
class ListSearch<SearchType, TypeList<Head, Tail> >
{
    enum { tmp = ListSearch<SearchType, Tail>::index } ;
    public:
        enum { index = tmp == -1 ? -1 : 1 + tmp };
};
```

Assuming all required headers have been included, the following example shows how `ListSearch` can be used:

```
int main()
{
    std::cout << ListSearch<char, TYPELIST_2(int, char)>::index << "\n";
}
```

22.6.3 Selecting from a TypeList

Next the selection of a type from a typelist given its index will be discussed. This is the inverse operation from obtaining the index of a 'searchtype', as covered by section 22.6.2.

Rather than defining an `enum` value, the current algorithm should define a type equal to the type at a given index position. If the type does not exist, the typedef can be made a synonym of `NullType` since `NullType` cannot appear in a typelist.

The following algorithm is used (the implementation of the parts is provided immediately following the descriptions of the algorithm's steps):

- The foundation of the algorithm is provided by a declaration of a struct `TypeAt`, expecting an index and a typelist:

```
template <int index, typename Typelist>
struct TypeAt;
```

- If the typelist equals `NullType` define the return type as `NullType` as well:

```
template <int index>
struct TypeAt<index, NullType>
{
    typedef NullType Type;
};
```

- If the search index equals 0, define the return type as the typelist's head:

```
template <typename Head, typename Tail>
struct TypeAt<0, TypeList<Head, Tail> >
{
    typedef Head Type;
};
```

- Otherwise, define the return type as the return type of the type at offset index - 1 in the typelist's tail. Note the `typename` following `typedef`: it is required as the defining type's `result` type is a nested type:

```
template <int index, typename Head, typename Tail>
```

```
struct TypeAt<index, TypeList<Head, Tail> >
{
    typedef typename TypeAt<index - 1, Tail>::Type Type;
};
```

Assuming all required headers have been included, the following example shows how `ListSearch` can be used:

```
int main()
{
    typedef TYPELIST_3(int, char, bool) list3;
    enum { test = 2 };

    std::cout <<
        (ListSearch<TypeAt<test, list3>::result, list3>::index == -1 ?
            "Illegal Index\n"
        :
            "Index represents a valid type\n");
}
```

22.6.4 Appending to a TypeList

The question of how to add an element to a typelist is handled using the same rule of thumb as used for answering the previous questions: design a recursive algorithm and implement the recursion through specializations.

To append a new type to a typelist, the following algorithm can be used:

- The basic template is a struct expecting a typelist and a type to add to the typelist:

  ```
  template <typename TypeList, typename NewType>
  struct Append;
  ```

- Adding `NullType`:
  - If the type to add is `NullType`, and the original type is `NullType`, the result itself is the `NullType`:

    ```
    template <>
    struct Append<NullType, NullType>
    {
        typedef NullType TList;
    };
    ```

    Note that the simple alternative:

    ```
    template <typename TypeList>
    struct Append<TypeList, NullType>
    {
        typedef TypeList Result;
    };
    ```

    is not a good idea, as it will match all types that are offered as the template's first template type parameter. E.g., `Append<int, NullType>` would be accepted, but would certainly not result in a `TypeList`.
  - When attempting to append `NullType` to an existing `TypeList`, leave the `TypeList` as-is:

    ```
    template <typename Head, typename Tail>
    struct Append<TypeList<Head, Tail>, NullType>
    {
        typedef TypeList<Head, Tail>  TList;
    };
    ```

- Appending other types than `NullType`:
  - If the typelist itself is `NullType`, the final typelist consists of the typelist containing the new type:

```
template <typename NewType>
struct Append<NullType, NewType>
{
    typedef TYPELIST_1(NewType) TList;
};
```

  - Otherwise, the final typelist consists of the head of the initial typelist and the typelist resulting from appending the new type to the intial typelist's tail:

```
template <typename Head, typename Tail, typename NewType>
struct Append<TypeList<Head, Tail>, NewType>
{
    typedef TypeList<Head, typename Append<Tail, NewType>::TList>
            TList;
};
```

    Once again: note that `typename` is required in front of `Append` (cf. section 22.1.1)

22.6.5 Erasing from a TypeList

The opposite from adding, erasing can simply be accomplished as well. Here is the algorithm, erasing the first occurrence of a type to erase from a typelist:

- The basic template is a struct expecting a typelist and a type to erase from the typelist:

```
template <typename TypeList, typename EraseType>
struct Erase;
```

- If the typelist itself is `NullType`, there's nothing to erase, and `NullType` is the result:

```
template <typename EraseType>
struct Erase<NullType, EraseType>
{
    typedef NullType Result;
};
```

- If the typelist's head equals the type to erase, then the result is the typelist's tail:

```
template <typename EraseType, typename Tail>
struct Erase<TypeList<EraseType, Tail>, EraseType>
{
    typedef Tail Result;
};
```

- Otherwise, the result is the typelist's head and the result obtained afdter erasing the type to be erased from the typelist's tail:

```
template <typename Head, typename Tail, typename EraseType>
struct Erase<TypeList<Head, Tail>, EraseType>
{
    typedef TypeList<Head,
                     typename Erase<Tail, EraseType>::Result> Result;
};
//
```

```
//ERASEALL
    template <typename TypeList, typename EraseType>
    struct EraseAll: public Erase<TypeList, EraseType>
    {};
```

But there's more: what if the intention is to erase all elements from the typelist? In that case also apply `Erase` to the tail when the type to erase matches the typelist's head. E.g., a `struct EraseAll` can be defined similarly to `Erase`, except for the case where the typelist's head matches the type to be removed: in that case `EraseAll` must also be applied to the typelist's tail, as there may be additional types to be removed in the tail as well.

Since `EraseAll` closely resembles `Erase`, let's see how we can use class derivation in combination with specializations to our benefit.

- First step: note that all alternatives of `Erase` but one can be used unaltered by `EraseAll`. So, `EraseAll` inherits from `Erase`, using the generic struct's template parameters:

  ```
  template <typename TypeList, typename EraseType>
  struct EraseAll: public Erase<TypeList, EraseType>
  {};
  ```

  Thus, `EraseAll` is a mere copy of `Erase`, and it could be used as a synonym of `Erase` (of course, erasing only the first element).

- Second (and last) step: define a specialization of `EraseAll` for the case where the type to be removed equals the typelist's head:

  ```
  template <typename EraseType, typename Tail>
  struct EraseAll<TypeList<EraseType, Tail>, EraseType>
  {
      typedef typename EraseAll<Tail, EraseType>::Result Result;
  };
  ```

  Note the `EraseAll` action on the template's tail.

The above two `EraseAll` definitions are all it takes to create a template that will do the job of erasing all occurrences of a type from a typelist, borrowing most of its code from the already existing `Erase` template. The effect of `EraseAll` vs. `Erase` can be seen when defining either `Erase` or `EraseAll` in the following example:

```
#include <iostream>
#include "erase.h"
#include "listsize.h"

    // change Erase to EraseAll to erase all 'int' types below
#define ERASE Erase

int main()
{
    std::cout <<
        ListSize<
            ERASE<TYPELIST_3(int, double, int), int>::Result
        >::size << "\n";
}
```

22.6.5.1 Erasing duplicates

So, erasing a type from a typelist can be accomplished. To remove duplicates all 'head' elements must be erased from a typelist's tail. To accomplish this, the following algorithm is used, defining the `EraseDuplicates` template:

- First, the general `EraseDuplicates` struct is declared. It expects as its single template type a `TypeList`:

```
template <typename TypeList>
struct EraseDuplicates;
```

- If the typelist is actually a `NullType`, we're done. The result will remain to be a `NullType`:

```
template <>
struct EraseDuplicates<NullType>
{
    typedef NullType  Result;
};
```

- Next, an `EraseDuplicates` specialization is defined for a true `Typelist`:

```
template <typename Head, typename Tail>
struct EraseDuplicates<TypeList<Head, Tail> >
...
```

  This specialization implements the following reasoning:

  - If `EraseDuplicates` erases duplicates, then no duplicates will be encountered in the typelist's tail if `EraseDuplicates` is recursively processes the template's tail. When this recursive call is completed, a typelist is obtained (called, e.g., `UniqueTail`) not containing any type duplicates in the typelist's tail:

```
typedef typename EraseDuplicates<Tail>::Result UniqueTail;
```

  - Of course, the above operation only processed the original typelist's tail. It may still contain duplicate's of the original template's head type. Since that latter type is already there (*viz.* at the original typelist's head) it can simply be removed from `UniqueTail`, producing the typelist `NewTail`:

```
typedef typename Erase<UniqueTail, Head>::Result  NewTail;
```

  - Finally, the resulting typelist is the original typelist's head and `NewTail`:

```
typedef TypeList<Head, NewTail>  Result;
```

Here's the full definition of `EraseDuplicates`, not exposing `UniequeTail` and `NewTail` for cosmetic reasons:

```
template <typename TypeList>
struct EraseDuplicates;

template <>
struct EraseDuplicates<NullType>
{
    typedef NullType  Result;
};

template <typename Head, typename Tail>
class EraseDuplicates<TypeList<Head, Tail> >
{
    typedef typename EraseDuplicates<Tail>::Result UniqueTail;
    typedef typename Erase<UniqueTail, Head>::Result  NewTail;

    public:
        typedef TypeList<Head, NewTail>  Result;
};
```

22.7 Using a TypeList

In the previous sections the definition and some of the features of typelists have been discussed. Most **C++** programmers consider typelists both exciting and an intellectual chalenge, honing their skills in the area of recursive programming.

Fortunately, there's more to typelist than a mere intellectual challenge. In this section of the chapter on Advanced Template Applications the following topics will be covered:

- Creating classes from a typelist
  Here the aim is to construct a new class consisting of instantiations of an existing basic template for each of the types mentined in a provided typelist;
- Accessing data members from the thus constructed conglomerate class by index, rather than name;
- `Tuples`, defining `structs` from typelists, having index-accessible data members for each of the types specified in a typelist.

Again, much (and more) of the materials covered below is found in Alexandrescu's (2001) book. Hopefully the current section is an tasty appetizer for the main courses offered by Andrei Alexandrescu.

22.7.1 The Wrap and GenScat templates

In this section the template class `GenScat` will be developed. The purpose of `GenScat` is to *create* a new class using on the one hand a basic building block of the class that's finally constructed and on the other hand a series of types that will be fed to the building block.

The building block itself is provided as a template template parameter, and the final class will inherit from all building blocks instantiated for each of the types specified in a provided typelist. However, there is a flaw in this plan.

If the typelist contains two types, say `int` and `double` and the building block class is `std::vector`, then the final `GenScat` class will inherit from `vector<int>` and `vector<double>`. There's nothing wrong with that. But what if the typelist contains two `int` type specifications? In that case the `GenScat` class will inherit from *two* `vector<int>` classes, and, e.g., `vector<int>::size()` will cause an ambiguity which is hard to solve. Alexandrescu (2001) in this regard writes (p.67):

> *There is one major source of annoyance...: you cannot use it when you have duplicate types in your typelist.*
> *.... There is no easy way to solve the ambiguity, [as the eventually derived class/FBB] ends up inheriting [the same base class/FBB] twice.*

It is true that the same base class is inherited multiple times when the typelist contains duplicate types, but there is a way around this problem. If instead of inheriting from the plain base classes these base classes would themselves be wrapped in unique classes, then these unique classes can be used to access the base classes by implication: since they are mere wrappers they inherit the functionality of the 'true' base classes.

Thus, the problem is shifted from type duplication to finding unique wrappers. Of course, *that* problem has been solved in principle in section 22.2.1.1, where wrappers around plain `int` values were introduced. A comparable wrapper can be designed in the context of class derivation. E.g.,

```
template <typename Base, int idx>
struct Wrap: public Base
{
    Wrap(Base const &base)
```

```
        :
            Base(base)
        {}
        Wrap()
        {}
    };
```

Using `Wrap` two `vector<int>` classes can be distinguished easily: `Wrap<1, vector<int> >` could be used to refer to one of the vectors, `Wrap<2, vector<int> >` could refer to the other vector. By ensuring that the index values never collide all wrapper types will be unique.

Uniqueness of the `Wrap` values is implemented by the `GenScat` class: it is itself a wrapper around the class `GenScatter`, that will do all the work. `GenScat` merely *seeds* `GenScatter` with an initial value:

```
template <typename Type, template <typename> class TemplateClass>
class GenScat: public GenScatter<Type, TemplateClass, 0>
{};
```

22.7.2 The GenScatter template

The interesting part of the exercise is of course the class template `GenScatter`. In its intended form (which actually turns out to be a specialization) `GenScatter` takes a typelist, a template class and a index value that is used to ensure uniqueness of the `Wrap` types.

With these ingredients, `GenScatter` creates a (fairly complex) class, that itself is normally derived from several `GenScatter` base classes. Each `GenScatter` class is eventually derived from a base class which is a `Wrap` class around the template class instantiated for each of the types in the provided typelist.

This complex arrangement itself causes yet another problem: it would be nice if the top-level derived class could be initialized by base class initializers. However, with normal class derivation indirect base classes cannot be initialized using base class initializers. This is a severe problem: if indirect base classes cannot be initialized by a `GenScat` class or by a class derived from `GenScat`, the types in the provided typelist cannot be refence types, as references *must* be initialized at construction time.

However, an indirect base class *can* be initialized by a top level derived class if it is a *virtual* base class (cf. section 14.4.2). The consequence of a virtual base class is that any duplicates of a virtual base class, following different paths in a class hierarchy will be merged into one class.

In the `GenScat` hierarchy the `Wrap` template class wrappers are the final (usable) base classes of the hierarchy. By defining these `Wrap` base classes as *virtual* base classes they *can* be initialized by `GenScat` (or by a class that itself is derived from `GenScat`), while *merging* of the `Wrap` base classes is prevented due to the fact that all `Wrap` base classes are unique.

It's time for some code. The `GenScatter` class performs the following tasks:

- It creates a class hierarchy, having unique `Wrap` template classes at its final nodes;
- It creates a typelist containing all `Wrap` base class types in the order in which they were constructed, to allow clients to obtain the `Wrap` template class wrapper matching a specific type in the provided typelist.

An illustration showing the layout of the final `GenScatter` class hierarchy and its subclasses is provided in figure 22.1.

The core definition of `GenScatter` expects a typelist, a template class and an index:

```
template <
    typename Head, typename Tail,
```

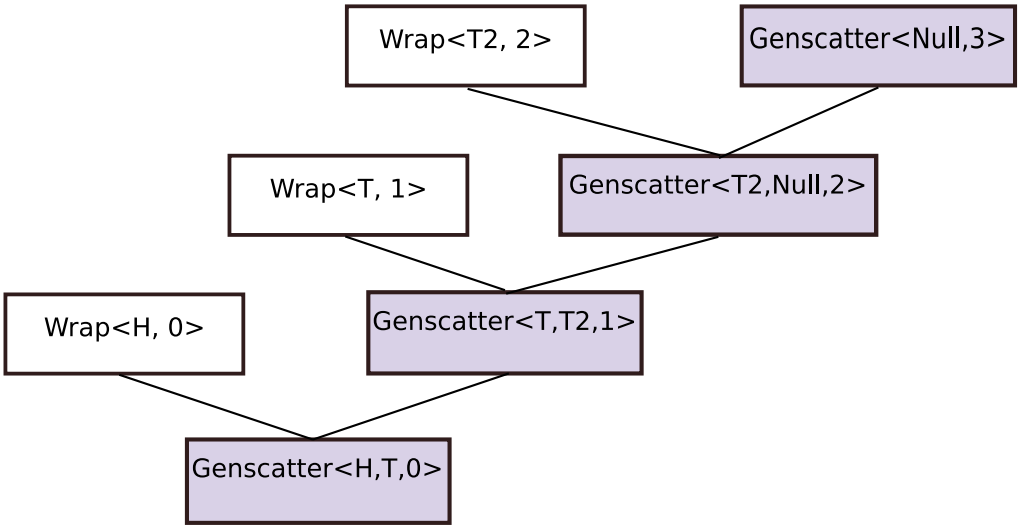

Figure 22.1: Layout of a GenScatter class hierarchy

```
    template <typename> class TemplateClass, int idx
>
class GenScatter<TypeList<Head, Tail>, TemplateClass, idx>
:
    virtual public Wrap<TemplateClass<Head>, idx>,
    public GenScatter<Tail, TemplateClass, idx + 1>
{
    typedef typename GenScatter<Tail, TemplateClass, idx + 1>::WrapList
            BaseWrapList;
    public:
        typedef TypeList<Wrap<TemplateClass<Head>, idx>, BaseWrapList>
                WrapList;
};
```

- Since the typelist's head is a plain type, it can immediately be used to define a `TemplateClass` type, which itself is wrapped in a `Wrap` template class wrapper using the unique index value that was passed to `GenScatter`.
- The `Wrap` template class wrapper is then defined as a *virtual* base class.
- A second hierarchal line starts at the typelist's tail, at the same time incrementing the index, thus ensuring that the next `Wrap` class will receive the next index value.
- At the end of the class definition the `WrapList` type is defined as the typelist consisting of the current `Wrap` wrapper as its head, and the base `GenScatter` class's `WrapList` as its tail.

`GenScatter`'s main template definition expects a simple type as its first template parameter. Since this is a plain type, the class can immediately define a virtual `Wrap` template class wrapper as its base class, and it can immediately define the `WrapList` type as a typelist containing the class's base class:

```
template <typename Type, template <typename> class TemplateClass, int idx>
class GenScatter
:
    virtual public Wrap<TemplateClass<Type>, idx>
{
```

```
        typedef Wrap<TemplateClass<Type>, idx> Base;

        public:
            typedef TYPELIST_1(Base)    WrapList;
    };
```

Finally, a specialization to handle the ending `NullType` is required: it merely defines an empty `WrapList`:

```
    template <template <typename> class TemplateClass, int idx>
    class GenScatter<NullType, TemplateClass, idx>
    {
        public:
            typedef NullType WrapList;
    };
```

Both the `Wrap` template class wrapper and the `GenScatter` class can normally be defined in the anonymous namespace, as they are only used at file-scope, by themselves, by `GenScatter` and by the occasional additional support functions and classes.

22.7.3 Support struct and function

Since the `GenScatter` class returns a typelist containing all `Wrap` base classes matching the types in the order in which they appeared in `GenScat`'s typelist, it is attractive to be able to obtain these `Wrap` base class types by their index numbers. Being able to reach these types by their indices allows, e.g., base class intializations as well as quick access to their respective members.

The class template `BaseClass` was designed with these thoughts in mind. It uses `AtIndex` (cf. section 22.6.3) to obtain a particular `Wrap` base class and returns the latter type as its `Type` type definition:

```
template <int idx, typename Derived>
struct BaseClass
{
    typedef typename TypeAt<idx, typename Derived::WrapList>::Type Type;
};
```

Once a `GenScatter` object is available, the following function template can be used to obtain a cast-less reference to any of its `Wrap` policy classes, given its index. Since the index can be matched one-to-one with the `GenScatter`'s typelist, clients should have no problem finding the appropriate index values for a particular problem at hand:

```
template <int idx, typename Derived>
struct BaseClass
{
    typedef typename TypeAt<idx, typename Derived::WrapList>::Type Type;
};
```

22.7.4 Using GenScatter

The class template `GenScat` can be used by itself, to define a simple `struct` containg various data members. The foundation of such a conglomerate `struct` could be the following `struct Field`:

```
    template <typename Type>
    struct Field
    {
```

```
        Type field;

        Field(Type v = Type())
        :
            field(v)
        {}
    };
```

Such an instant `struct` could be useful in various situations; due to the nature of the `struct Field`, all data types would by default be initialized to their natural defaults. E.g., `GenScat` can be used directly as follows:

```
    GenScat<TYPELIST_2(int, int), Field> gs;

    base<1>(gs).d_value = 12;
    cout << base<0>(gs).d_value << " " << base<1>(gs).d_value << endl;
```

The above code, when it is run from `main()` will write the values 0 and 12, showing that default initialization and assignment to the individual fields is simply implemented.

Useful as this may be, sometimes more refined initializations may be necessary. E.g, an application needs a `struct` having two `int` data fields and a reference to a `std::string`. Since the `struct` contains a reference field, an initialization is required at construction time. In this case a `struct` can be derived from `GenScat`, while providing a constructor for the derived class performing the necessary intiializations. For situations like these, the `BaseClass` support struct (section 22.7.3) comes in quite handy. Here is the `struct MyStruct`, derived from the appropriate `GenScat` template, including its field-initializations:

```
    struct MyStruct: public
                        GenScat<TYPELIST_3(int, std::string &, int), Field>
    {
        MyStruct(int i, std::string &text, int i2)
        :
            BaseClass<0, MyStruct>::Type(i),
            BaseClass<1, MyStruct>::Type(text),
            BaseClass<2, MyStruct>::Type(i2)
         {}
    };
```

Note how each of the types in the provided typelist has its order-number mapped to the index used with the `BaseClass` invocations. Also, since `MyStruct` is also an object of its base class (`GenScat`), it can be specified as the `Derived` argument of `BaseClass`. Furthermore, from the types specified in the typelist the types of acceptable arguments of the `Type`s to be initialized can be derived. E.g., the `string text` is passed as argument to `Type` when initializing the second field.

The following example shows how `MyStruct` can be used:

```
    string text("hello");
    MyStruct myStruct(12345, text, 12);

    cout << base<0>(myStruct).field << " " <<
            base<1>(myStruct).field << " " <<
            base<2>(myStruct).field << endl;

    base<0>(myStruct).field = 123;
    base<1>(myStruct).field = "new text";

    cout << base<0>(myStruct).field << "\n" <<
            "'text' now contains: " << text << endl;
```

When these lines of code are placed in a `main()` function, and the program is run the following output is produced showing proper initialization, reassignment and reassignment of the refered to `string text` via the appropriate `MyStruct` field:

```
    12345 hello 12
    123
    'text' now contains: new text
```

As a final example consider the `struct Vectors`:

```
struct Vectors: public GenScat<TYPELIST_3(int, std::string, int), std::vector>
{
    Vectors()
    :
        BaseClass<0, Vectors>::Type(std::vector<int>(1)),
        BaseClass<1, Vectors>::Type(std::vector<std::string>(2)),
        BaseClass<2, Vectors>::Type(std::vector<int>(3))
     {}
};
```

The `struct Vectors` uses `std::vector` as its template template parameter, and `Vectors` objects will thus offer three `std::vectors`: the first containing `int`s, the second `string`s, and the third again `int`s. Due to the nature of the `Wrap` template class wrapper, the three `std::vector` base classes of `Vectors` must be initialized by `std::vector` objects, and the constructor simply provides three vectors of varying sizes. Alternatively, the constructor could be furnished with three `vector` references or three `size_t` values to allow a more flexible initialization.

A `Vectors` object could be used as follows, showing that the `base` support function (cf. section 22.7.3) provides easy access to the `vector` base class of choice:

```
    Vectors vects;

    cout << base<0>(vects).size() << " " << base<1>(vects).size() << " " <<
            base<2>(vects).size() << endl;
```

Running this code fragment produces the output '`1 2 3`', as expected.

Chapter 23

Concrete Examples Of C++

In this chapter several concrete examples of **C++** programs, classes and templates will be presented. Topics covered by this document such as virtual functions, `static` members, etc. are illustrated in this chapter. The examples roughly follow the organization of earlier chapters.

First, examples using `stream` classes are presented, including some detailed examples illustrating polymorphism. With the advent of the ANSI/ISO standard, classes supporting streams based on *file descriptors* are no longer available, including the Gnu `procbuf` extension. These classes were frequently used in older **C++** programs. This section of the **C++** Annotations develops an alternative: classes extending `streambuf`, allowing the use of file descriptors, and classes around the `fork()` system call.

Finally, we'll touch the subjects of scanner and parser generators, and show how these tools may be used in **C++** programs. These final examples assume a certain familiarity with the concepts underlying these tools, like grammars, parse-trees and parse-tree decoration. Once the input for a program exceeds a certain level of complexity, it's advantageous to use scanner- and parser-generators to produce code doing the actual input recognition. One of the examples in this chapter describes the usage of these tools in a **C++** environment.

23.1 Distinguishing lvalues from rvalues with operator[]()

(ISN, see concrete/lvalues/lvalues.cc)

23.2 Using file descriptors with 'streambuf' classes

23.2.1 Classes for output operations

Extensions to the ANSI/ISO standard may be available allowing us to read from and/or write to *file descriptors*. However, such extensions are not standard, and may thus vary or be unavailable across compilers and/or compiler versions. On the other hand, a file descriptor can be considered a device. So it seems natural to use the class `streambuf` as the starting point for constructing classes interfacing file descriptors.

In this section we will construct classes which may be used to write to a device identified by a file descriptor: it may be a file, but it could also be a pipe or socket. Section 23.2.2 discusses reading from devices given their file descriptors, while section 23.4.1 reconsiders redirection, discussed earlier in section 6.6.1.

Basically, deriving a class for output operations is simple. The only member function that *must* be overridden is the `virtual` member `int overflow(int c)`. This member is responsible for writing

characters to the device once the class's buffer is full. If `fd` is a file descriptor to which information may be written, and if we decide against using a buffer then the member `overflow()` can simply be:

```
class UnbufferedFD: public std::streambuf
{
    public:
        int overflow(int c);
        ...
};

int UnbufferedFD::overflow(int c)
{
    if (c != EOF)
    {
        if (write(d_fd, &c, 1) != 1)
            return EOF;
    }
    return c;
}
```

The argument received by `overflow()` is either written as a value of type `char` to the file descriptor, or `EOF` is returned.

This simple function does not use an output buffer. As the use of a buffer is strongly advised (see also the next section), the construction of a class using an output buffer will be discussed next in somewhat greater detail.

When an output buffer is used, the `overflow()` member will be a bit more complex, as it is now only called when the buffer is full. Once the buffer is full, we *first* have to flush the buffer, for which the (virtual) function `streambuf::sync()` is available. Since `sync()` is a virtual function, classes derived from `std::streambuf` may redefine `sync()` to flush a buffer `std::streambuf` itself doesn't know about.

Overriding `sync()` and using it in `overflow()` is not all that has to be done: eventually we might have less information than fits into the buffer. So, at the end of the lifetime of our special `streambuf` object, its buffer might only be partially full. Therefore, we must make sure that the buffer is flushed once our object goes out of scope. This is of course very simple: `sync()` should be called by the destructor as well.

Now that we've considered the consequences of using an output buffer, we're almost ready to construct our derived class. We will add a couple of additional features, though.

- First, we should allow the user of the class to specify the size of the output buffer.
- Second, it should be possible to construct an object of our class before the file descriptor is actually known. Later, in section 23.4 we'll encounter a situation where this feature will be used.

In order to save some space, the successful operation of the various functions was not checked. In 'real life' implementations these checks should of course not be omitted. Our class `ofdnstreambuf` has the following characteristics:

- The class itself is derived from `std::streambuf`. It defines three data members, keeping track of the size of the buffer, the file descriptor and the buffer itself. Here is the full class interface

```
class ofdnstreambuf: public std::streambuf
{
    size_t d_bufsize;
    int     d_fd;
    char    *d_buffer;
```

```
    public:
        ofdnstreambuf();
        ofdnstreambuf(int fd, size_t bufsize = 1);
        ~ofdnstreambuf();
        void open(int fd, size_t bufsize = 1);
        int sync();
        int overflow(int c);
};
```

- Its default constructor merely initializes the buffer to 0. Slightly more interesting is its constructor expecting a filedescriptor and a buffer size: it simply passes its arguments on to the class's `open()` member (see below). Here are the constructors:

```
inline ofdnstreambuf::ofdnstreambuf()
:
    d_bufsize(0),
    d_buffer(0)
{}

inline ofdnstreambuf::ofdnstreambuf(int fd, size_t bufsize)
{
    open(fd, bufsize);
}
```

- The destructor calls the overridden function `sync()`, writing any characters stored in the output buffer to the device. If there's no buffer, the destructor needs to perform no actions:

```
inline ofdnstreambuf::~ofdnstreambuf()
{
    if (d_buffer)
    {
        sync();
        delete[] d_buffer;
    }
}
```

  Even though the device is not closed in the above implementation this may not always be what one wants. It is left as an exercise to the reader to change this class in such a way that the device may optionally remain open. This approach was followed in, e.g., the `Bobcat library`[1]. See also section 23.2.2.2.

- The `open()` member initializes the buffer. Using `setp()`, the begin and end points of the buffer are set. This is used by the `streambuf` base class to initialize `pbase()`, `pptr()`, and `epptr()`:

```
inline void ofdnstreambuf::open(int fd, size_t bufsize)
{
    d_fd = fd;
    d_bufsize = bufsize == 0 ? 1 : bufsize;

    d_buffer = new char[d_bufsize];
    setp(d_buffer, d_buffer + d_bufsize);
}
```

- The member `sync()` will flush the as yet unflushed contents of the buffer to the device. Next, the buffer is reinitialized using `setp()`. Note that `sync()` returns 0 after a successful flush operation:

```
inline int ofdnstreambuf::sync()
```

[1] http://bobcat.sourceforge.net

```
    {
        if (pptr() > pbase())
        {
            write(d_fd, d_buffer, pptr() - pbase());
            setp(d_buffer, d_buffer + d_bufsize);
        }
        return 0;
    }
```

- Finally, the member `overflow()` is overridden. Since this member is called from the `streambuf` base class when the buffer is full, `sync()` is called first to flush the filled up buffer to the device. As this recreates an empty buffer, the character `c` which could not be written to the buffer by the `streambuf` base class is now entered into the buffer using the member functions `pptr()` and `pbump()`. Notice that entering a character into the buffer is implemented using available `streambuf` member functions, rather than doing it 'by hand', which might invalidate `streambuf`'s internal bookkeeping:

```
    inline int ofdnstreambuf::overflow(int c)
    {
        sync();
        if (c != EOF)
        {
            *pptr() = c;
            pbump(1);
        }
        return c;
    }
```

- The member function implementations use low-level functions to operate on the file descriptors. So apart from `streambuf` the header file `unistd.h` must have been read by the compiler before the implementations of the member functions can be compiled.

Depending on the *number* of arguments, the following program uses the `ofdstreambuf` class to copy its standard input to file descriptor `STDOUT_FILENO`, which is the symbolic name of the file descriptor used for the standard output. Here is the program:

```
#include <string>
#include <iostream>
#include <istream>
#include "fdout.h"
using namespace std;

int main(int argc)
{
    ofdnstreambuf   fds(STDOUT_FILENO, 500);
    ostream         os(&fds);

    switch (argc)
    {
        case 1:
            os << "COPYING cin LINE BY LINE\n";
            for (string  s; getline(cin, s); )
                os << s << endl;
        break;

        case 2:
            os << "COPYING cin BY EXTRACTING TO os.rdbuf()\n";

            cin >> os.rdbuf();      // Alternatively, use:  cin >> &fds;
```

```
            break;

            case 3:
                os << "COPYING cin BY INSERTING cin.rdbuf() into os\n";
                os << cin.rdbuf();
            break;
        }
    }
```

23.2.2 Classes for input operations

When classes to be used for input operation are derived from `std::streambuf`, they should be provided with an input buffer of at least one character. The one-character input buffer allows for the use of the member functions `istream::putback()` or `istream::ungetc()`. Stream classes (like `istream`) normally allow us to unget at least one character using their member functions `putback()` or `ungetc()`. This is important, as these stream classes usually interface to `streambuf` objects. Although it is strictly speaking not necessary to implement a buffer in classes derived from `streambuf` using buffers in these cases is strongly advised: the implementation is very simple and straightforward, and the applicability of such classes will be greatly improved. Therefore, in all our classes derived from the class `streambuf` *at least* a buffer of one character will be defined.

23.2.2.1 Using a one-character buffer

When deriving a class (e.g., `ifdstreambuf`) from `streambuf` using a buffer of one character, at least its member `streambuf::underflow()` should be overridden, as this is the member to which all requests for input are eventually directed. Since a buffer is also needed, the member `streambuf::setg()` is used to inform the `streambuf` base class of the size of the input buffer, so that it is able to set up its input buffer pointers correctly. This will ensure that `eback()`, `gptr()`, and `egptr()` return correct values.

The required class shows the following characteristics:

- Like the class designed for output operations, this class is derived from `std::streambuf` as well. The class defines two data members, one of them a fixed-sized one character buffer. The data members are defined as `protected` data members so that derived classes (e.g., see section 23.2.2.3) can access them. Here is the full class interface:

```
class ifdstreambuf: public std::streambuf
{
    protected:
        int     d_fd;
        char    d_buffer[1];
    public:
        ifdstreambuf(int fd);
        int underflow();
};
```

- The constructor initializes the buffer. However, this initialization is done so that `gptr()` will be equal to `egptr()`. Since this implies that the buffer is empty, `underflow()` will immediately be called to refill the buffer:

```
inline ifdstreambuf::ifdstreambuf(int fd)
:
    d_fd(fd)
{
    setg(d_buffer, d_buffer + 1, d_buffer + 1);
}
```

- Finally `underflow()` is overridden. It will first ensure that the buffer is really empty. If not, then the next character in the buffer is returned. If the buffer is really empty, it is refilled by reading from the file descriptor. If this fails (for whatever reason), `EOF` is returned. More sophisticated implementations could react more intelligently here, of course. If the buffer could be refilled, `setg()` is called to set up `streambuf`'s buffer pointers correctly:

```
inline int ifdstreambuf::underflow()
{
    if (gptr() < egptr())
        return *gptr();

    if (read(d_fd, d_buffer, 1) <= 0)
        return EOF;

    setg(d_buffer, d_buffer, d_buffer + 1);
    return *gptr();
}
```

- The implementations of the member functions use low-level functions to operate the file descriptors, so apart from `streambuf` the header file `unistd.h` must have been read by the compiler before the implementations of the member functions can be compiled.

This completes the construction of the `ifdstreambuf` class. It is used in the following program:

```
#include <iostream>
#include <istream>
#include <unistd.h>
#include "ifdbuf.h"
using namespace std;

int main(int argc)
{
    ifdstreambuf fds(STDIN_FILENO);
    istream      is(&fds);

    cout << is.rdbuf();
}
```

23.2.2.2 Using an n-character buffer

How complex would things get if we would decide to use a buffer of substantial size? Not that complex. The following class allows us to specify the size of a buffer, but apart from that it is basically the same class as `ifdstreambuf` developed in the previous section. To make things a bit more interesting, in the class `ifdnstreambuf` developed here, the member `streambuf::xsgetn()` is also overridden, to optimize reading of series of characters. Furthermore, a default constructor is provided which can be used in combination with the `open()` member to construct an `istream` object before the file descriptor becomes available. Then, once the descriptor becomes available, the `open()` member can be used to initiate the object's buffer. Later, in section 23.4, we'll encounter such a situation.

To save some space, the success of various calls was not checked. In 'real life' implementations, these checks should, of course, not be omitted. The class `ifdnstreambuf` has the following characteristics:

- Once again, it is derived from `std::streambuf`: Like the class `ifdstreambuf` (section 23.2.2.1), its data members are protected. Since the buffer's size is configurable, this size is kept in a dedicated data member, `d_bufsize`:

```
class ifdnstreambuf: public std::streambuf
```

```
{
    protected:
        int         d_fd;
        size_t    d_bufsize;
        char*       d_buffer;
    public:
        ifdnstreambuf();
        ifdnstreambuf(int fd, size_t bufsize = 1);
        ~ifdnstreambuf();
        void open(int fd, size_t bufsize = 1);
        int underflow();
        std::streamsize xsgetn(char *dest, std::streamsize n);
};
```

- The default constructor does not allocate a buffer, and can be used to construct an object before the file descriptor becomes known. A second constructor simply passes its arguments to `open()` which will then initialize the object so that it can actually be used:

```
inline ifdnstreambuf::ifdnstreambuf()
:
    d_bufsize(0),
    d_buffer(0)
{}
inline ifdnstreambuf::ifdnstreambuf(int fd, size_t bufsize)
{
    open(fd, bufsize);
}
```

- If the object has been initialized by `open()`, its destructor will both delete the object's buffer and use the file descriptor to close the device:

```
ifdnstreambuf::~ifdnstreambuf()
{
    if (d_bufsize)
    {
        close(d_fd);
        delete[] d_buffer;
    }
}
```

  Even though the device is closed in the above implementation this may not always be what one wants. In cases where the open file descriptor is already available the intention may be to use that descriptor repeatedly, each time using a newly constructed `ifdnstreambuf` object. It is left as an exercise to the reader to change this class in such a way that the device may optionally be closed. This approach was followed in, e.g., the `Bobcat library`[2].

- The `open()` member simply allocates the object's buffer. It is assumed that the calling program has already opened the device. Once the buffer has been allocated, the base class member `setg()` is used to ensure that `eback()`, `gptr()`, and `egptr()` return correct values:

```
void ifdnstreambuf::open(int fd, size_t bufsize)
{
    d_fd = fd;
    d_bufsize = bufsize;
    d_buffer = new char[d_bufsize];
    setg(d_buffer, d_buffer + d_bufsize, d_buffer + d_bufsize);
}
```

[2] http://bobcat.sourceforge.net

- The overridden member `underflow()` is implemented almost identically to `ifdstreambuf`'s (section 23.2.2.1) member. The only difference is that the current class supports a buffer of larger sizes. Therefore, more characters (up to `d_bufsize`) may be read from the device at once:

```
int ifdnstreambuf::underflow()
{
    if (gptr() < egptr())
        return *gptr();

    int nread = read(d_fd, d_buffer, d_bufsize);

    if (nread <= 0)
        return EOF;

    setg(d_buffer, d_buffer, d_buffer + nread);
    return *gptr();
}
```

- Finally `xsgetn()` is overridden. In a loop, `n` is reduced until 0, at which point the function terminates. Alternatively, the member returns if `underflow()` fails to obtain more characters. This member optimizes the reading of series of characters: instead of calling `streambuf::sbumpc()` `n` times, a block of `avail` characters is copied to the destination, using `streambuf::gpumb()` to consume `avail` characters from the buffer using one function call:

```
std::streamsize ifdnstreambuf::xsgetn(char *dest, std::streamsize n)
{
    int nread = 0;

    while (n)
    {
        if (!in_avail())
        {
            if (underflow() == EOF)
                break;
        }

        int avail = in_avail();

        if (avail > n)
            avail = n;

        memcpy(dest + nread, gptr(), avail);
        gbump(avail);

        nread += avail;
        n -= avail;
    }

    return nread;
}
```

- The implementations of the member functions use low-level functions to operate the file descriptors. So apart from `streambuf` the header file `unistd.h` must have been read by the compiler before the implementations of the member functions can be compiled.

The member function `xsgetn()` is called by `streambuf::sgetn()`, which is a `streambuf` member. The following example illustrates the use of this member function with a `ifdnstreambuf` object:

```
#include <unistd.h>
```

```
#include <iostream>
#include <istream>
#include "ifdnbuf.h"
using namespace std;

int main(int argc)
{
                                // internally: 30 char buffer
    ifdnstreambuf fds(STDIN_FILENO, 30);

    char buf[80];               // main() reads blocks of 80
                                // chars
    while (true)
    {
        size_t n = fds.sgetn(buf, 80);
        if (n == 0)
            break;
        cout.write(buf, n);
    }
}
```

23.2.2.3 Seeking positions in 'streambuf' objects

When devices support *seek operations*, classes derived from `streambuf` should override the members `streambuf::seekoff()` and `streambuf::seekpos()`. The class `ifdseek`, developed in this section, can be used to read information from devices supporting such seek operations. The class `ifdseek` was derived from `ifdstreambuf`, so it uses a character buffer of just one character. The facilities to perform seek operations, which are added to our new class `ifdseek`, will make sure that the input buffer is reset when a seek operation is requested. The class could also be derived from the class `ifdnstreambuf`; in which case, the arguments to reset the input buffer must be adapted in such a way that its second and third parameters point beyond the available input buffer. Let's have a look at the characteristics of `ifdseek`:

- As mentioned, `ifdseek` is derived from `ifdstreambuf`. Like the latter class, `ifdseek`'s member functions use facilities declared in `unistd.h`. So, the compiler must have seen `unistd.h` before it can compile the class's members functions. To reduce the amount of typing when specifying types and constants from `std::streambuf` and `std::ios`, several `typedef`s are defined at the class's very top. These typedefs refer to types that are defined in the header file `ios`, which must therefore be included as well before the compiler reads `ifdseek`'s class definition. Here is the class's interface:

```
class ifdseek: public ifdstreambuf
{
    typedef std::streambuf::pos_type        pos_type;
    typedef std::streambuf::off_type        off_type;
    typedef std::ios::seekdir               seekdir;
    typedef std::ios::openmode              openmode;

    public:
        ifdseek(int fd);
        pos_type seekoff(off_type offset, seekdir dir, openmode);
        pos_type seekpos(pos_type offset, openmode mode);
};
```

- The class is given a rather basic implementation. The only required constructor expects the device's file descriptor. It has no special tasks to perform and only needs to call its base class constructor:

```
inline ifdseek::ifdseek(int fd)
```

```
    :
        ifdstreambuf(fd)
    {}
```

- The member `seek_off()` is responsible for performing the actual seek operations. It calls `lseek()` to seek a new position in a device whose file descriptor is known. If seeking succeeds, `setg()` is called to define an already empty buffer, so that the base class's `underflow()` member will refill the buffer at the next input request.

```
    ifdseek::pos_type ifdseek::seekoff(off_type off, seekdir dir, openmode)
    {
        pos_type pos =
            lseek
            (
                d_fd, off,
                (dir ==  std::ios::beg) ? SEEK_SET :
                (dir ==  std::ios::cur) ? SEEK_CUR :
                                          SEEK_END
            );

        if (pos < 0)
            return -1;

        setg(d_buffer, d_buffer + 1, d_buffer + 1);
        return pos;
    }
```

- Finally, the companion function `seekpos` is overridden as well: it is actually defined as a call to `seekoff()`:

```
    inline ifdseek::pos_type ifdseek::seekpos(pos_type off, openmode mode)
    {
        return seekoff(off, std::ios::beg, mode);
    }
```

An example of a program using the class `ifdseek` is the following. If this program is given its own source file using input redirection then seeking is supported, and with the exception of the first line, every other line is shown twice:

```
#include "fdinseek.h"
#include <string>
#include <iostream>
#include <istream>
#include <iomanip>
using namespace std;

int main(int argc)
{
    ifdseek fds(0);
    istream is(&fds);
    string  s;

    while (true)
    {
        if (!getline(is, s))
            break;

        streampos pos = is.tellg();
```

```
            cout << setw(5) << pos << ": '" << s << "'\n";

            if (!getline(is, s))
                break;

            streampos pos2 = is.tellg();

            cout << setw(5) << pos2 << ": '" << s << "'\n";

            if (!is.seekg(pos))
            {
                cout << "Seek failed\n";
                break;
            }
        }
    }
```

23.2.2.4 Multiple 'unget()' calls in 'streambuf' objects

As mentioned before, `streambuf` classes and classes derived from `streambuf` should support *at least* ungetting the last read character. Special care must be taken when *series* of `unget()` calls must be supported. In this section the construction of a class supporting a configurable number of `istream::unget()` or `istream::putback()` calls is discussed.

Support for multiple (say 'n') `unget()` calls is implemented by reserving an initial section of the input buffer, which is gradually filled up to contain the last `n` characters read. The class was implemented as follows:

- Once again, the class is derived from `std::streambuf`. It defines several data members, allowing the class to perform the bookkeeping required to maintain an unget-buffer of a configurable size:

```
    class fdunget: public std::streambuf
    {
        int         d_fd;
        size_t    d_bufsize;
        size_t    d_reserved;
        char*       d_buffer;
        char*       d_base;

        public:
            fdunget(int fd, size_t bufsz, size_t unget);
            ~fdunget();
            int underflow();
    };
```

- The class's constructor expects a file descriptor, a buffer size and the number of characters that can be ungot or pushed back as its arguments. This number determines the size of a *reserved* area, defined as the first `d_reserved` bytes of the class's input buffer.
  - The input buffer will always be at least one byte larger than `d_reserved`. So, a certain number of bytes may be read. Then, once `reserved` bytes have been read at least `reserved` bytes can be ungot.
  - Next, the starting point for reading operations is configured: it is called `d_base`, pointing to a location `reserved` bytes from the start of `d_buffer`. This will always be the point where the buffer refills start.
  - Now that the buffer has been constructed, we're ready to define `streambuf`'s buffer pointers using `setg()`. As no characters have been read yet, all pointers are set to point to `d_base`.

If `unget()` is called at this point, no characters are available, so `unget()` will (correctly) fail.

- Eventually, the refill buffer's size is determined as the number of allocated bytes minus the size of the reserved area.

Here is the class's constructor:

```
fdunget::fdunget(int fd, size_t bufsz, size_t unget)
:
    d_fd(fd),
    d_reserved(unget)
{
    size_t allocate =
            bufsz > d_reserved ?
                bufsz
            :
                d_reserved + 1;

    d_buffer = new char[allocate];

    d_base = d_buffer + d_reserved;
    setg(d_base, d_base, d_base);

    d_bufsize = allocate - d_reserved;
}
```

- The class's destructor simply returns the memory allocated for the buffer to the common pool:

```
inline fdunget::~fdunget()
{
    delete[] d_buffer;
}
```

- Finally, `underflow()` is overridden.
  - Firstly, the standard check to determine whether the buffer is really empty is applied.
  - If empty, it determines the number of characters that could potentially be ungot. At this point, the input buffer is exhausted. So this value may be any value between 0 (the initial state) or the input buffer's size (when the reserved area has been filled up completely, and all current characters in the remaining section of the buffer have also been read).
  - Next the number of bytes to move into the reserved area is computed. This number is at most `d_reserved`, but it is equal to the actual number of characters that can be ungot if this value is smaller.
  - Now that the number of characters to move into the reserved area is known, this number of characters is moved from the input buffer's end to the area immediately before `d_base`.
  - Then the buffer is refilled. This all is standard, but notice that reading starts from `d_base` and not from `d_buffer`.
  - Finally, `streambuf`'s read buffer pointers are set up. `Eback()` is set to `move` locations before `d_base`, thus defining the guaranteed unget-area, `gptr()` is set to `d_base`, since that's the location of the first read character after a refill, and `egptr()` is set just beyond the location of the last character read into the buffer.

Here is `underflow()`'s implementation:

```
int fdunget::underflow()
{
    if (gptr() < egptr())
        return *gptr();
```

```
        size_t ungetsize = gptr() - eback();
        size_t move = std::min(ungetsize, d_reserved);

        memcpy(d_base - move, egptr() - move, move);

        int nread = read(d_fd, d_base, d_bufsize);
        if (nread <= 0)       // none read -> return EOF
            return EOF;

        setg(d_base - move, d_base, d_base + nread);

        return *gptr();
    }
```

The following program illustrates the `class fdunget`. It reads at most 10 characters from the standard input, stopping at `EOF`. A guaranteed unget-buffer of 2 characters is defined in a buffer holding 3 characters. Just before reading a character, the program tries to unget at most 6 characters. This is, of course, not possible; but the program will nicely unget as many characters as possible, considering the actual number of characters read:

```
#include "fdunget.h"
#include <string>
#include <iostream>
#include <istream>
using namespace std;

int main(int argc)
{
    fdunget fds(0, 3, 2);
    istream is(&fds);
    char    c;

    for (int idx = 0; idx < 10; ++idx)
    {
        cout << "after reading " << idx << " characters:\n";
        for (int ug = 0; ug <= 6; ++ug)
        {
            if (!is.unget())
            {
                cout
                << "\tunget failed at attempt " << (ug + 1) << "\n"
                << "\trereading: '";

                is.clear();
                while (ug--)
                {
                    is.get(c);
                    cout << c;
                }
                cout << "'\n";
                break;
            }
        }

        if (!is.get(c))
        {
            cout << " reached\n";
            break;
        }
```

```
            cout << "Next character: " << c << endl;
        }
    }
    /*
        Generated output after 'echo abcde | program':

        after reading 0 characters:
                unget failed at attempt 1
                rereading: ''
        Next character: a
        after reading 1 characters:
                unget failed at attempt 2
                rereading: 'a'
        Next character: b
        after reading 2 characters:
                unget failed at attempt 3
                rereading: 'ab'
        Next character: c
        after reading 3 characters:
                unget failed at attempt 4
                rereading: 'abc'
        Next character: d
        after reading 4 characters:
                unget failed at attempt 4
                rereading: 'bcd'
        Next character: e
        after reading 5 characters:
                unget failed at attempt 4
                rereading: 'cde'
        Next character:

        after reading 6 characters:
                unget failed at attempt 4
                rereading: 'de
        '
         reached
    */
```

23.3 Fixed-sized field extraction from istream objects

Usually when extracting information from `istream` objects `operator>>`, the standard extraction operator, is perfectly suited for the task as in most cases the extracted fields are white-space or otherwise clearly separated from each other. But this does not hold true in all situations. For example, when a web-form is posted to some processing script or program, the receiving program may receive the form field's values as *url-encoded* characters: letters and digits are sent unaltered, blanks are sent as + characters, and all other characters start with % followed by the character's ascii-value represented by its two digit hexadecimal value.

When decoding url-encoded information, a simple hexadecimal extraction won't work, since that will extract as many hexadecimal characters as available, instead of just two. Since the letters `a-f` and `0-9` are legal hexadecimal characters, a text like `My name is 'Ed'`, url-encoded as

```
My+name+is+%60Ed%27
```

will result in the extraction of the hexadecimal values `60ed` and `27`, instead of `60` and `27`. The name `Ed` will disappear from view, which is clearly not what we want.

In this case, having seen the %, we could extract 2 characters, put them in an `istringstream` object, and extract the hexadecimal value from the `istringstream` object. A bit cumbersome, but doable. Other approaches, however, are possible as well.

The following class `fistream` for *fixed-sized field istream* defines an `istream` class supporting both fixed-sized field extractions and blank-delimited extractions (as well as unformatted `read()` calls). The class may be initialized as a *wrapper* around an existing `istream`, or it can be initialized using the name of an existing file. The class is derived from `istream`, allowing all extractions and operations supported by `istream`s in general. The class will need the following data members:

- `d_filebuf`: a filebuffer used when `fistream` reads its information from a named (existing) file. Since the filebuffer is only needed in that case, and since it must be allocated dynamically, it is defined as an `auto_ptr<filebuf>` object.
- `d_streambuf`: a pointer to `fistream`'s `streambuf`. It will point to `filebuf` when `fistream` opens a file by name. When an existing `istream` is used to construct an `fistream`, it will point to the existing `istream`'s `streambuf`.
- `d_iss`: an `istringstream` object which is used for the fixed field extractions.
- `d_width`: a `size_t` indicating the width of the field to extract. If 0 no fixed field extractions will be used, but information will be extracted from the `istream` base class object using standard extractions.

Here is the initial section of `fistream`'s class interface:

```
class fistream: public std::istream
{
    std::auto_ptr<std::filebuf> d_filebuf;
    std::streambuf *d_streambuf;
    std::istringstream d_iss;
    size_t d_width;
```

As mentioned, `fistream` objects can be constructed from either a filename or an existing `istream` object. Thus, the class interface shows two constructors:

```
        fistream(std::istream &stream);
        fistream(char const *name,
            std::ios::openmode mode = std::ios::in);
```

When an `fistream` object is constructed using an existing `istream` object, the `fistream`'s `istream` part will simply use the `stream`'s `streambuf` object:

```
fistream::fistream(istream &stream)
:
    istream(stream.rdbuf()),
    d_streambuf(rdbuf()),
    d_width(0)
{}
```

When an `fstream` object is constructed using a filename, the `istream` base initializer is given a new `filebuf` object to be used as its `streambuf`. Since the class's data members are not initialized before the class's base class has been constructed, `d_filebuf` can only be initialized thereafter. By then, the `filebuf` is only available as `rdbuf()`, which returns a `streambuf`. However, as it is actually a `filebuf`, a `reinterpret_cast` is used to cast the `streambuf` pointer returned by `rdbuf()` to a `filebuf *`, so `d_filebuf` can be initialized:

```
fistream::fistream(char const *name, ios::openmode mode)
```

```
:
    istream(new filebuf()),
    d_filebuf(reinterpret_cast<filebuf *>(rdbuf())),
    d_streambuf(d_filebuf.get()),
    d_width(0)
{
    d_filebuf->open(name, mode);
}
```

There is only one additional public member: `setField(field const &)`. This member is used to define the size of the next field to extract. Its parameter is a reference to a `field` class, a *manipulator class* defining the width of the next field.

Since a `field &` is mentioned in `fistream`'s interface, `field` must be declared before `fistream`'s interface starts. The class `field` itself is simple: it declares `fistream` as its friend, and it has two data members: `d_width` specifies the width of the next field, `d_newWidth` is set to `true` if `d_width`'s value should actually be used. If `d_newWidth` is false, `fistream` will return to its standard extraction mode. The class `field` furthermore has two constructors: a default constructor, setting `d_newWidth` to `false` and a second constructor expecting the width of the next field to extract as its value. Here is the class `field`:

```
    class field
    {
        friend class fistream;
        size_t d_width;
        bool     d_newWidth;

        public:
            field(size_t width);
            field();
    };

    inline field::field(size_t width)
    :
        d_width(width),
        d_newWidth(true)
    {}

    inline field::field()
    :
        d_newWidth(false)
    {}
```

Since `field` declares `fistream` as its friend, `setField` may inspect `field`'s members directly.

Time to return to `setField()`. This function expects a reference to a `field` object, initialized in either of three different ways:

- `field()`: When `setField()`'s argument is a `field` object constructed by its default constructor the next extraction will use the same fieldwidth as the previous extraction.
- `field(0)`: When this `field` object is used as `setField()`'s argument, fixed-sized field extraction stops, and the `fistream` will act like any standard `istream` object.
- `field(x)`: When the `field` object itself is initialized by a non-zero size_t value `x`, then the next field width will be `x` characters wide. The preparation of such a field is left to `setBuffer()`, `fistream`'s only private member.

Here is `setField()`'s implementation:

```
std::istream &fistream::setField(field const &params)
{
    if (params.d_newWidth)                  // new field size requested
        d_width = params.d_width;           // set new width

    if (!d_width)                           // no width?
        rdbuf(d_streambuf);                 // return to the old buffer
    else
        setBuffer();                        // define the extraction buffer

    return *this;
}
```

The private member `setBuffer()` defines a buffer of `d_width + 1` characters, and uses `read()` to fill the buffer with `d_width` characters. The buffer is terminated by an `ASCII-Z` character. This buffer is then used to initialize the `d_str` member. Finally, `fistream`'s `rdbuf()` member is used to extract the `d_str`'s data via the `fistream` object itself:

```
void fistream::setBuffer()
{
    char *buffer = new char[d_width + 1];

    rdbuf(d_streambuf);                     // use istream's buffer to
    buffer[read(buffer, d_width).gcount()] = 0; // read d_width chars,
                                                // terminated by ascii-Z
    d_iss.str(buffer);
    delete buffer;

    rdbuf(d_iss.rdbuf());                   // switch buffers
}
```

Although `setField()` could be used to configure `fistream` to use or not to use fixed-sized field extraction using manipulators is probably preferable. To allow `field` objects to be used as manipulators, an overloaded extraction operator was defined, accepting an `istream &` and a `field const &` object. Using this extraction operator, statements like

```
fis >> field(2) >> x >> field(0);
```

are possible (assuming `fis` is a `fistream` object). Here is the overloaded `operator>>`, as well as its declaration:

```
istream &std::operator>>(istream &str, field const &params)
{
    return reinterpret_cast<fistream *>(&str)->setField(params);
}
```

Declaration:

```
namespace std
{
    istream &operator>>(istream &str, FBB::field const &params);
}
```

Finally, an example. The following program uses a `fistream` object to url-decode url-encoded information appearing at its standard input:

```
    int main()
```

```
    {
        fistream fis(cin);

        fis >> hex;
        while (true)
        {
            size_t x;
            switch (x = fis.get())
            {
                case '\n':
                    cout << endl;
                break;
                case '+':
                    cout << ' ';
                break;
                case '%':
                    fis >> field(2) >> x >> field(0);
                // FALLING THROUGH
                default:
                    cout << static_cast<char>(x);
                break;
                case EOF:
                return 0;
            }
        }
    }
    /*
        Generated output after:
            echo My+name+is+%60Ed%27 | a.out

        My name is 'Ed'
    */
```

23.4 The 'fork()' system call

From the **C** programming language, the `fork()` system call is well known. When a program needs to start a new process, `system()` can be used, but this requires the program to wait for the *child process* to terminate. The more general way to spawn subprocesses is to call `fork()`.

In this section we will see how **C++** can be used to wrap classes around a complex system call like `fork()`. Much of what follows in this section directly applies to the Unix operating system, and the discussion will therefore focus on that operating system. However, other systems usually provide comparable facilities. The following discussion is based heavily on the notion of *design patterns* (cf. *Gamma et al.* (1995) Design Patterns, Addison-Wesley)

When `fork()` is called, the current program is duplicated in memory, thus creating a new process, and both processes continue their execution just below the `fork()` system call. The two processes may, however, inspect the return value of `fork()`: the return value in the original process (called the *parent process*) differs from the return value in the newly created process (called the *child process*):

- In the *parent process* `fork()` returns the *process ID* of the child process created by the `fork()` system call. This is a positive integer value.
- In the *child process* `fork()` returns 0.
- If `fork()` fails, -1 is returned.

A basic `Fork` class should hide all bookkeeping details of a system call like `fork()` from its users.

The class Fork developed here will do just that. The class itself only needs to take care of the proper execution of the fork() system call. Normally, fork() is called to start a child process, usually boiling down to the execution of a separate process. This child process may expect input at its standard input stream and/or may generate output to its standard output and/or standard error streams. Fork does not know all this, and does not have to know what the child process will do. However, Fork objects should be able to activate their child processes.

Unfortunately, Fork's constructor cannot know what actions its child process should perform. Similarly, it cannot know what actions the parent process should perform. For this particular situation, the *template method design pattern* was developed. According to Gamma c.s., the *template method design pattern*

> "Define(s) the skeleton of an algorithm in an operation, deferring some steps to subclasses. (The) Template Method (design pattern) lets subclasses redefine certain steps of an algorithm, without changing the algorithm's structure."

This design pattern allows us to define an *abstract base class* already providing the essential steps related to the fork() system call and deferring the implementation of certain normally used parts of the fork() system call to subclasses.

The Fork abstract base class itself has the following characteristics:

- It defines a data member d_pid. This data member will contain the child's *process id* (in the parent process) and the value 0 in the child process. Its public interface declares but two members:
  - a fork() member function, performing the actual forking (i.e., it will create the (new) child process);
  - an *empty* virtual destructor ~Fork(), which will be overridden by derived classes defining their own destructors.

    ```
    inline Fork::~Fork()
    {}
    ```

  Here is Fork's interface:

    ```
    class Fork
    {
        int d_pid;

        public:
            virtual ~Fork();
            void fork();

        protected:
            int pid() const;
            virtual void childRedirections();
            virtual void parentRedirections();

            virtual void childProcess() = 0;    // both MUST be implemented
            virtual void parentProcess() = 0;

            int waitForChild();                 // returns the status
    };
    ```

- All remaining member functions are declared in the class's protected section and can thus *only* be used by derived classes. They are:
  - The member function pid(), allowing derived classes to access the system fork()'s return value:

    ```
    inline int Fork::pid() const
    ```

```
{
    return d_pid;
}
```

- A member `int waitForChild()`, which can be called by parent processes to wait for the completion of their child processes (as discussed below). This member is declared in the class interface. Its implementation is:

```
#include "fork.ih"

int Fork::waitForChild()
{
    int status;

    waitpid(d_pid, &status, 0);

    return WEXITSTATUS(status);
}
```

  This simple implementation returns the child's *exit status* to the parent. The called system function `waitpid()` *blocks* until the child terminates.

- When `fork()` system calls are used, *parent processes* and *child processes* must always be distinguished. The main distinction between these processes is that `d_pid` will be equal to the child's process-id in the parent process, while `d_pid` will be equal to 0 in the child process itself. Since these two processes must always be distinguished (and present), their implementation by classes derived from `Fork` is enforced by `Fork`'s interface: the members `childProcess()`, defining the child process' actions and `parentProcess()`, defining the parent process' actions were defined as pure virtual functions.

- In addition, communication between parent- and child processes may use standard streams or other facilities, like *pipes* (cf. section 23.4.3). To facilitate this inter-process communication, derived classes *may* implement:

  * `childRedirections()`: this member should be implemented if any standard stream (`cin, cout`) or `cerr` must be redirected in the *child* process (cf. section 23.4.1);
  * `parentRedirections()`: this member should be implemented if any standard stream (`cin, cout`) or `cerr` must be redirected in the *parent* process.

  Redirection of the standard streams will be necessary if parent- and child processes should communicate with each other via the standard streams. Here are their default definitions provided by the class's interface:

```
inline void Fork::childRedirections()
{}
inline void Fork::parentRedirections()
{}
```

The member function `fork()` calls the system function `fork()` (Caution: since the system function `fork()` is called by a member function having the same name, the `::` scope resolution operator must be used to prevent a recursive call of the member function itself). After calling `::fork()`, depending on its return value, either `parentProcess()` or `childProcess()` is called. Maybe redirection is necessary. `Fork::fork()`'s implementation calls `childRedirections()` just before calling `childProcess()`, and `parentRedirections()` just before calling `parentProcess()`:

```
#include "fork.ih"

void Fork::fork()
{
    if ((d_pid = ::fork()) < 0)
        throw "Fork::fork() failed";

    if (d_pid == 0)                 // childprocess has pid == 0
```

```
        {
            childRedirections();
            childProcess();

            exit(1);                    // we shouldn't come here:
                                        // childProcess() should exit
        }

        parentRedirections();
        parentProcess();
    }
```

In `fork.cc` the class's *internal header file* `fork.ih` is included. This header file takes care of the inclusion of the necessary system header files, as well as the inclusion of `fork.h` itself. Its implementation is:

```
#include "fork.h"
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

Child processes should not return: once they have completed their tasks, they should terminate. This happens automatically when the child process performs a call to a member of the `exec...()` family, but if the child itself remains active, then it must make sure that it terminates properly. A child process normally uses `exit()` to terminate itself, but note that `exit()` prevents the activation of destructors of objects defined at the same or more superficial nesting levels than the level at which `exit()` is called. Destructors of globally defined objects *are* activated when `exit()` is used. When using `exit()` to terminate `childProcess()`, it should either itself call a support member function defining all nested objects it needs, or it should define all its objects in a compound statement (e.g., using a `throw` block) calling `exit()` beyond the compound statement.

Parent processes should normally wait for their children to complete. The terminating child processes inform their parent that they are about to terminate by sending out a *signal* which should be caught by their parents. If child processes terminate and their parent processes do not catch those signal then such child processes remain visible as so-called *zombie* processes.

If parent processes must wait for their children to complete, they may call the member `waitForChild()`. This member returns the exit status of a child process to its parent.

There exists a situation where the *child* process *continues* to live, but the *parent* dies. In nature this happens all the time: parents tend to die before their children do. In our context (i.e. **C++**), this is called a *daemon* program: the parent process dies and the child program continues to run as a child of the basic `init` process. Again, when the child eventually dies a signal is sent to its 'step-parent' `init`. No zombie is created here, as `init` catches the termination signals of all its (step-) children. The construction of a daemon process is very simple, given the availability of the class `Fork` (cf. section 23.4.2).

23.4.1 Redirection revisited

Earlier, in section 6.6.1, it was noted that within a **C++** program, streams could be redirected using the `ios::rdbuf()` member function. By assigning the `streambuf` of a stream to another stream, both stream objects access the same `streambuf`, thus implementing redirection at the level of the programming language itself.

Note that this is fine within the context of the **C++** program, but if that context is left, the redirection terminates, as the operating system does not know about `streambuf` objects. This happens, e.g., when a program uses a `system()` call to start a subprogram. The program at the end of this section uses **C++** redirection to redirect the information inserted into `cout` to a file, and then calls

```
system("echo hello world")
```

to echo a well-known line of text. Since `echo` writes its information to the standard output, this would be the program's redirected file if **C++**'s redirection would be recognized by the operating system.

Actually, this doesn't happen; and `hello world` still appears at the program's standard output instead of the redirected file. A solution of this problem involves redirection at the operating system level, for which some operating systems (e.g., Unix and friends) provide system calls like `dup()` and `dup2()`. Examples of these system calls are given in section 23.4.3.

Here is the example of the *failing redirection* at the system level following **C++** redirection using `streambuf` redirection:

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace::std;

int main()
{
    ofstream of("outfile");

    cout.rdbuf(of.rdbuf());
    cout << "To the of stream" << endl;
    system("echo hello world");
    cout << "To the of stream" << endl;
}
/*
    Generated output: on the file 'outfile'

    To the of stream
    To the of stream

    On standard output:

    hello world
*/
```

23.4.2 The 'Daemon' program

Applications exist in which the only purpose of `fork()` is to start a child process. The parent process terminates immediately after spawning the child process. If this happens, the child process continues to run as a child process of `init`, the always running first process on Unix systems. Such a process is often called a *daemon*, running as a background process.

Although the following example can easily be constructed as a plain **C** program, it was included in the **C++** Annotations because it is so closely related to the current discussion of the `Fork` class. I thought about adding a `daemon()` member to that class, but eventually decided against it because the construction of a daemon program is very simple and requires no features other than those currently offered by the class `Fork`. Here is an example illustrating the construction of a daemon program:

```
#include <iostream>
#include <unistd.h>
#include "fork.h"

class Daemon: public Fork
{
    public:
```

```
        virtual void parentProcess()        // the parent does nothing.
        {}

        virtual void childProcess()
        {
            sleep(3);                       // actions taken by the child
                                            // just a message...
            std::cout << "Hello from the child process\n";
            exit (0);                       // The child process exits.
        }
};

int main()
{
    Daemon daemon;

    daemon.fork();          // program immediately returns
    return 0;
}

/*
    Generated output:
The next command prompt, then after 3 seconds:
Hello from the child process
*/
```

23.4.3 The class 'Pipe'

Redirection at the system level involves the use of *file descriptors*, created by the `pipe()` system call. When two processes want to communicate using such file descriptors, the following takes place:

- The process constructs two *associated file descriptors* using the `pipe()` system call. One of the file descriptors is used for writing, the other file descriptor is used for reading.
- Forking takes place (i.e., the system `fork()` function is called), duplicating the file descriptors. Now we have four file descriptors as both the child process and the parent process have their own copies of the two file descriptors created by `pipe()`.
- One process (say, the parent process) will use the filedescriptors for *reading*. It should close its filedescriptor intended for *writing*.
- The other process (say, the child process) will use the filedescriptors for *writing*. It should close its filedescriptor intended for *reading*.
- All information written by the child process to the file descriptor intended for writing, can now be read by the parent process from the corresponding file descriptor intended for reading, thus establishing a communication channel between the child- and the parent process.

Though basically simple, errors may easily creep in: purposes of file descriptors available to the two processes (child- or parent-) may easily get mixed up. To prevent bookkeeping errors, the bookkeeping may be properly set up once, to be hidden therafter inside a class like the `Pipe` class constructed here. Let's have a look at its characteristics (before the implementations can be compiled, the compiler must have read the class's header file as well as the file `unistd.h`):

- The `pipe()` system call expects a pointer to two `int` values, which will represent, respectively, the file descriptors to use for accessing the *reading end* and the *writing end* of the constructed pipe, after `pipe()`'s successful completion. To avoid confusion, an `enum` is defined associating

these ends with symbolic constants. Furthermore, the class stores the two file descriptors in a data member `d_fd`. Here is the class header and its private data:

```
class Pipe
{
    enum    RW { READ, WRITE };
    int     d_fd[2];
```

- The class only needs a default constructor. This constructor calls `pipe()` to create a set of associated file descriptors used for accessing both ends of a pipe:

```
Pipe::Pipe()
{
    if (pipe(d_fd))
        throw "Pipe::Pipe(): pipe() failed";
}
```

- The members `readOnly()` and `readFrom()` are used to configure the pipe's reading end. The latter function is used to set up redirection, by providing an alternate file descriptor which can be used to read from the pipe. Usually this alternate file descriptor is `STDIN_FILENO`, allowing `cin` to extract information from the pipe. The former function is merely used to configure the reading end of the pipe: it closes the matching writing end, and returns a file descriptor that can be used to read from the pipe:

```
int Pipe::readOnly()
{
    close(d_fd[WRITE]);
    return d_fd[READ];
}
void Pipe::readFrom(int fd)
{
    readOnly();

    redirect(d_fd[READ], fd);
    close(d_fd[READ]);
}
```

- `writeOnly()` and two `writtenBy()` members are available to configure the writing end of a pipe. The former function is merely used to configure the writing end of the pipe: it closes the matching reading end, and returns a file descriptor that can be used to write to the pipe:

```
int Pipe::writeOnly()
{
    close(d_fd[READ]);
    return d_fd[WRITE];
}
void Pipe::writtenBy(int fd)
{
    writtenBy(&fd, 1);
}
void Pipe::writtenBy(int const *fd, size_t n)
{
    writeOnly();

    for (size_t idx = 0; idx < n; idx++)
        redirect(d_fd[WRITE], fd[idx]);

    close(d_fd[WRITE]);
}
```

For the latter member two overloaded versions are available:

- `writtenBy(int fileDescriptor)` is used to configure *single* redirection, so that a specific file descriptor (usually `STDOUT_FILENO` or `STDERR_FILENO`) may be used to write to the pipe;
- `(writtenBy(int *fileDescriptor, size_t n = 2))` may be used to configure *multiple* redirection, providing an array argument containing file descriptors. Information written to any of these file descriptors is actually written into the pipe.

- The class has one private data member, `redirect()`, which is used to define a redirection using the `dup2()` system call. This function expects two file descriptors. The first file descriptor represents a file descriptor which can be used to access the device's information, the second file descriptor is an alternate file descriptor which may also be used to access the device's information once `dup2()` has completed successfully. Here is `redirect()`'s implementation:

```
void Pipe::redirect(int d_fd, int alternateFd)
{
    if (dup2(d_fd, alternateFd) < 0)
        throw "Pipe: redirection failed";
}
```

Now that redirection can be configured easily using one or more `Pipe` objects, we'll now use `Fork` and `Pipe` in several demonstration programs.

23.4.4 The class 'ParentSlurp'

The class `ParentSlurp`, derived from `Fork`, starts a child process which *execs* a program (like `/bin/ls`). The (standard) output of the execed program is then read by the parent process. The parent process will (for demonstration purposes) write the lines it receives to its standard output stream, while prepending linenumbers to the received lines. It is most convenient here to redirect the parents standard input stream, so that the parent can read the *output* from the child process from its `std::cin` *input* stream. Therefore, the only pipe that's used is used as an *input* pipe at the parent, and an *output* pipe at the child.

The class `ParentSlurp` has the following characteristics:

- It is derived from `Fork`. Before starting `ParentSlurp`'s class interface, the compiler must have read both `fork.h` and `pipe.h`. Furthermore, the class only uses one data member: a `Pipe` object `d_pipe`.
- Since `Pipe`'s constructor automatically constructs a pipe, and since `d_pipe` is automatically constructed by `ParentSlurp`'s default constructor, there is no need to define `ParentSlurp`'s constructor explicitly. As no construtor needs to be implemented, all `ParentSlurp`'s members can be declared as `protected` members. Here is the class's interface:

```
class ParentSlurp: public Fork
{
    Pipe    d_pipe;

    protected:
        virtual void childRedirections();
        virtual void parentRedirections();
        virtual void childProcess();
        virtual void parentProcess();
};
```

- The `childRedirections()` member configures the pipe as a pipe for reading. So, all information written to the child's standard output stream will end up in the pipe. The big advantage of this all is that no streams around file descriptors are needed to write to a file descriptor:

```
inline void ParentSlurp::childRedirections()
{
    d_pipe.writtenBy(STDOUT_FILENO);
}
```

- The `parentRedirections()` member, configures its end of the pipe as a reading pipe. It does so by redirecting the reading end of the pipe to its standard input file descriptor (`STDIN_FILENO`), thus allowing extractions from `cin` instead of using streams built around file descriptors.

```
inline void ParentSlurp::parentRedirections()
{
    d_pipe.readFrom(STDIN_FILENO);
}
```

- The `childProcess()` member only has to concentrate on its own actions. As it only needs to execute a program (writing information to its standard output), the member consists of but one statement:

```
inline void ParentSlurp::childProcess()
{
    execl("/bin/ls", "/bin/ls", static_cast<char *>(0));
}
```

- The `parentProcess()` member simply 'slurps' the information appearing at its standard input. Doing so, it actually reads the child's output. It copies the received lines to its standard output stream after having prefixed line numbers to them:

```
void ParentSlurp::parentProcess()
{
    std::string     line;
    size_t    nr = 1;

    while (getline(std::cin, line))
        std::cout << nr++ << ": " << line << std::endl;

    waitForChild();
}
```

The following program simply constructs a `ParentSlurp` object, and calls its `fork()` member. Its output consists of a numbered list of files in the directory where the program is started. Note that the program also needs the `fork.o, pipe.o` and `waitforchild.o` object files (see earlier sources):

```
int main()
{
    ParentSlurp ps;

    ps.fork();
    return 0;
}
/*
    Generated Output (example only, actually obtained output may differ):

    1: a.out
    2: bitand.h
    3: bitfunctional
    4: bitnot.h
    5: daemon.cc
    6: fdinseek.cc
    7: fdinseek.h
    ...
*/
```

23.4.5 Communicating with multiple children

The next step up the ladder is the construction of a child-process monitor. Here, the parent process is responsible for all its child processes, but it also must read their standard output. The user may enter information at the parent process' standard input, for which a simple *command language* is defined:

- `start` will start a new child process. The parent will return the ID (a number) to the user. The ID may thereupon be used to send a message to that particular child process
- `<nr> text` will send "`text`" to the child process having ID `<nr>`;
- `stop <nr>` will terminate the child process having ID `<nr>`;
- `exit` will terminate the parent as well as all of its children.

Furthermore, the child process that hasn't received text for some time will complain, by sending a message to the parent-process. The parent process will then simply transmit the received message to the user, by copying it to the standard output stream.

A problem with programs like our monitor is that these programs allow *asynchronous input* from multiple sources: input may appear at the standard input as well as at the input-sides of pipes. Also, multiple output channels are used. To handle situations like these, the `select()` system call was developed.

23.4.5.1 The class 'Select'

The `select()` system call was developed to handle asynchronous *I/O multiplexing*. This system call can be used to handle, e.g., input appearing simultaneously at a set of file descriptors.

The `select()` system function is rather complex, and its full discussion is beyond the **C++** Annotations' scope. However, its use may be simplified by providing a `class Selector`, hiding its details and offering an easy-to-use public interface. Here its characteristics are discussed:

- Most of `Select`'s members are very small, allowing us to define most of its members as inline functions. The class requires quite a few data members. Most of them of types that were specifically constructed for use by `select()`. Therefore, before the class interface can be handled by the compiler, various header files must have been read by it:

```
#include <limits.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
```

- The class definition and its data members may appear next. The data type `fd_set` is a type designed to be used by `select()` and variables of this type contain the set of filedescriptors on which `select()` has sensed some activity. Furthermore, `select()` allows us to fire an *asynchronous alarm*. To specify alarm times, the class receives a `timeval` data member. The remaining members are used by the class for internal bookkeeping purposes, illustrated below. Here is the class's interface:

```
class Selector
{
    fd_set          d_read;
    fd_set          d_write;
    fd_set          d_except;
    fd_set          d_ret_read;
    fd_set          d_ret_write;
    fd_set          d_ret_except;
```

```
        timeval         d_alarm;
        int             d_max;
        int             d_ret;
        int             d_readidx;
        int             d_writeidx;
        int             d_exceptidx;

        public:
            Selector();

            int wait();
            int nReady();
            int readFd();
            int writeFd();
            int exceptFd();
            void setAlarm(int sec, int usec = 0);
            void noAlarm();
            void addReadFd(int fd);
            void addWriteFd(int fd);
            void addExceptFd(int fd);
            void rmReadFd(int fd);
            void rmWriteFd(int fd);
            void rmExceptFd(int fd);

        private:
            int checkSet(int *index, fd_set &set);
            void addFd(fd_set *set, int fd);
    };
```

The following member functions are part of the class's public interface:

- `Selector()`: the (default) constructor. It clears the read, write, and execute `fd_set` variables, and switches off the alarm. Except for `d_max`, the remaining data members do not require initializations. Here is the implementation of `Selector`'s constructor:

```
Selector::Selector()
{
    FD_ZERO(&d_read);
    FD_ZERO(&d_write);
    FD_ZERO(&d_except);
    noAlarm();
    d_max = 0;
}
```

- `int wait()`: this member function will *block()* until activity is sensed at any of the file descriptors monitored by the `Selector` object, or if the *alarm* times out. It will throw an exception when the `select()` system call itself fails. Here is `wait()`'s implementation:

```
int Selector::wait()
{
    timeval t = d_alarm;

    d_ret_read = d_read;
    d_ret_write = d_write;
    d_ret_except = d_except;

    d_readidx = 0;
    d_writeidx = 0;
    d_exceptidx = 0;
```

```
    d_ret = select(d_max, &d_ret_read, &d_ret_write, &d_ret_except, &t);

    if (d_ret < 0)
        throw "Selector::wait()/select() failed";

    return d_ret;
}
```

- `int nReady`: this member function's return value is defined only when `wait()` has returned. In that case it returns 0 for a alarm-timeout, -1 if `select()` failed, and the number of file descriptors on which activity was sensed otherwise. It can be implemented inline:

```
inline int Selector::nReady()
{
    return d_ret;
}
```

- `int readFd()`: this member function's return value also is defined only after `wait()` has returned. Its return value is -1 if no (more) input file descriptors are available. Otherwise the next file descriptor available for reading is returned. Its inline implementation is:

```
inline int Selector::readFd()
{
    return checkSet(&d_readidx, d_ret_read);
}
```

- `int writeFd()`: operating analogously to `readFd()`, it returns the next file descriptor to which output is written. Using `d_writeidx` and `d_ret_read`, it is implemented analogously to `readFd()`;
- `int exceptFd()`: operating analogously to `readFd()`, it returns the next exception file descriptor on which activity was sensed. Using `d_except_idx` and `d_ret_except`, it is implemented analogously to `readFd()`;
- `void setAlarm(int sec, int usec = 0)`: this member activates `Select`'s alarm facility. At least the number of seconds to wait for the alarm to go off must be specified. It simply assigns values to `d_alarm`'s fields. Then, at the next `Select::wait()` call, the alarm will fire (i.e., `wait()` returns with return value 0) once the configured alarm-interval has passed. Here is its (inline) implementation:

```
inline void Selector::setAlarm(int sec, int usec)
{
    d_alarm.tv_sec  = sec;
    d_alarm.tv_usec = usec;
}
```

- `void noAlarm()`: this member switches off the alarm, by simply setting the alarm interval to a very long period. Implemented inline as:

```
inline void Selector::noAlarm()
{
    setAlarm(INT_MAX, INT_MAX);
}
```

- `void addReadFd(int fd)`: this member adds a file descriptor to the set of input file descriptors monitored by the `Selector` object. The member function `wait()` will return once input is available at the indicated file descriptor. Here is its inline implementation:

```
inline void Selector::addReadFd(int fd)
{
    addFd(&d_read, fd);
}
```

- `void addWriteFd(int fd)`: this member adds a file descriptor to the set of output file descriptors monitored by the `Selector` object. The member function `wait()` will return once output is available at the indicated file descriptor. Using `d_write`, it is implemented analogously as `addReadFd()`;
- `void addExceptFd(int fd)`: this member adds a file descriptor to the set of exception file descriptors to be monitored by the `Selector` object. The member function `wait()` will return once activity is sensed at the indicated file descriptor. Using `d_except`, it is implemented analogously as `addReadFd()`;
- `void rmReadFd(int fd)`: this member removes a file descriptor from the set of input file descriptors monitored by the `Selector` object. Here is its inline implementation:

```
inline void Selector::rmReadFd(int fd)
{
    FD_CLR(fd, &d_read);
}
```

- `void rmWriteFd(int fd)`: this member removes a file descriptor from the set of output file descriptors monitored by the `Selector` object. Using `d_write`, it is implemented analogously as `rmReadFd()`;
- `void rmExceptFd(int fd)`: this member removes a file descriptor from the set of exception file descriptors to be monitored by the `Selector` object. Using `d_except`, it is implemented analogously as `rmReadFd()`;

The class's remaining (two) members are support members, and should not be used by non-member functions. Therefore, they should be declared in the class's `private` section:

- The member `addFd()` adds a certain file descriptor to a certain `fd_set`. Here is its implementation:

```
void Selector::addFd(fd_set *set, int fd)
{
    FD_SET(fd, set);
    if (fd >= d_max)
        d_max = fd + 1;
}
```

- The member `checkSet()` tests whether a certain file descriptor (`*index`) is found in a certain `fd_set`. Here is its implementation:

```
int Selector::checkSet(int *index, fd_set &set)
{
    int &idx = *index;

    while (idx < d_max && !FD_ISSET(idx, &set))
        ++idx;

    return idx == d_max ? -1 : idx++;
}
```

23.4.5.2 The class 'Monitor'

The `monitor` program uses a `Monitor` object to do most of the work. The class has only one public constructor and one public member, `run()`, to perform its tasks. Therefore, all other member functions described below should be declared in the class's `private` section.

`Monitor` defines the `private enum Commands`, symbolically listing the various commands its input language supports, as well as several data members, among which a `Selector` object and a `map` using

child order numbers as its keys, and pointer to `Child` objects (see section 23.4.5.3) as its values. Furthermore, `Monitor` has a static array member `s_handler[]`, storing pointers to member functions handling user commands.

A destructor should have been implemented too, but its implementation is left as an exercise to the reader. Before the class interface can be processed by the compiler, it must have seen `select.h` and `child.h`. Here is the class header, including the interface of the nested function object class `Find`:

```
class Monitor
{
    enum Commands
    {
        UNKNOWN,
        START,
        EXIT,
        STOP,
        TEXT,
        sizeofCommands
    };

    class Find
    {
        int     d_nr;
        public:
            Find(int nr);
            bool operator()(std::map<int, Child *>::value_type &vt)
                                                                const;
    };

    Selector                d_selector;
    int                     d_nr;
    std::map<int, Child *>  d_child;

    static void (Monitor::*s_handler[])(int, std::string const &);

    public:
        enum Done
        {};

        Monitor();
        void run();

    private:
        static void killChild(std::map<int, Child *>::value_type it);
        static void initialize();

        Commands    next(int *value, std::string *line);
        void    processInput();
        void    processChild(int fd);

        void    createNewChild(int, std::string const &);
        void    exiting(int = 0, std::string const &msg = std::string());
        void    sendChild(int value, std::string const &line);
        void    stopChild(int value, std::string const &);
        void    unknown(int, std::string const &);
};
```

Since there's only one non-class type data member, the class's constructor remains very short and could be implemented inline. However, the array `s_handler`, storing pointers to functions needs to be

initialized as well. This can be accomplished in several ways:

- Since the `Command` enumeration only contains a fairly limited set of commands, compile-time initialization could be considered:

```
void (Monitor::*Monitor::s_handler[])(int, string const &)  =
{
    &Monitor::unknown,              // order follows enum Command's
    &Monitor::createNewChild,       // elements
    &Monitor::exiting,
    &Monitor::stopChild,
    &Monitor::sendChild,
};
```

  The advantage of this is that it's simple, and not requiring any run-time effort. The disadvantage is of course relatively complex maintenance. If for some reason `Commads` is modified, `s_handler` must be modified as well. In cases like these, compile-time initialization is a little bit asking for trouble. There is a simple alternative though, which admittedly does take some execution time:

- A static member may be called before the first `Monitor` object is constructed, which initializes the elements of the array explicitly. This has the advantage of robustness against reordering of enumeration values, which is important: enumerations *do* receive modifications during the development cycle of a class. Maintenance is still required if new values are added to the enumeration, but in that case maintenance is required anyway.

- Using a static member that's explicitly called from `main()` may become a burden, or may be considered unacceptable, as it puts an additional responsibility with the software engineer, rather than with the software. It's a matter of taste whether that's a consideration to take seriously or not. If the initialization function is not called, the program will clearly fail and repairing the error caused by not calling the initialization function is easily repaired. If that's considered bad practice, the initialization function may be called from the class constructors as well. The following initialization function used in the current implementation of the class `Monitor`:

```
void (Monitor::*Monitor::s_handler[sizeofCommands])(int, string const &);

void Monitor::initialize()
{
    if (s_handler[UNKNOWN] != 0)    // already initialized
        return;

    s_handler[UNKNOWN] =    &Monitor::unknown;
    s_handler[START] =      &Monitor::createNewChild;
    s_handler[EXIT] =       &Monitor::exiting;
    s_handler[STOP] =       &Monitor::stopChild;
    s_handler[TEXT] =       &Monitor::sendChild;
}
```

Since the initialization function immediately returns if the initialization has already been performed, `Monitor`'s constructor may call the initialization and still defensibly be implemented inline:

```
inline Monitor::Monitor()
:
    d_nr(0)
{
    initialize();
}
```

The core of `Monitor`'s activities are performed by `run()`. It performs the following tasks:

- Initially, the `Monitor` object only listens to its standard input: the set of input file descriptors to which `d_selector` will listen is initialized to `STDIN_FILENO`.

- Then, in a loop `d_selector`'s `wait()` function is called. If input on `cin` is available, it is processed by `processInput()`. Otherwise, the input has arived from a child process. Information sent by children is processed by `processChild()`.
- To prevent *zombies*, the child processes must catch *their* children's termination signals. This will be discussed below (In an earlier version `Monitor` caught the termination signals. As noted by Ben Simons (`ben at mrxfx dot com`) this is inappropriate: the process spawning child processes has that responsibility (so, the parent process is responsible for its child processes; a child process is in turn responsible for its own child processes). Thanks, Ben).

Here is `run()`'s implementation:

```
#include "monitor.ih"

void Monitor::run()
{
    d_selector.addReadFd(STDIN_FILENO);

    while (true)
    {
        cout << "? " << flush;
        try
        {
            d_selector.wait();

            int fd;
            while ((fd = d_selector.readFd()) != -1)
            {
                if (fd == STDIN_FILENO)
                    processInput();
                else
                    processChild(fd);
            }
            cout << "NEXT ...\n";

        }
        catch (char const *msg)
        {
            exiting(1, msg);
        }
    }
}
```

The member function `processInput()` reads the commands entered by the user via the program's standard input stream. The member itself is rather simple: it calls `next()` to obtain the next command entered by the user, and then calls the corresponding function using the matching element of the `s_handler[]` array. The members `processInput()` and `next()` were defined as follows:

```
void Monitor::processInput()
{
    string line;
    int    value;
    Commands cmd = next(&value, &line);
    (this->*s_handler[cmd])(value, line);
}

Monitor::Commands Monitor::next(int *value, string *line)
{
```

```
        if (!getline(cin, *line))
            exiting(1, "Command::next(): reading cin failed");

        if (*line == "start")
            return START;

        if (*line == "exit" || *line == "quit")
        {
            *value = 0;
            return EXIT;
        }

        if (line->find("stop") == 0)
        {
            istringstream istr(line->substr(4));
            istr >> *value;
            return !istr ? UNKNOWN : STOP;
        }

        istringstream istr(line->c_str());
        istr >> *value;
        if (istr)
        {
            getline(istr, *line);
            return TEXT;
        }

        return UNKNOWN;
    }
```

All other input sensed by `d_select` has been created by child processes. Because `d_select`'s `readFd()` member returns the corresponding input file descriptor, this descriptor can be passed to `processChild()`. Then using a `ifdstreambuf` (see section 23.2.2.1), its information is read from an input stream. The *communication protocol* used here is rather basic: To every line of input sent to a child, the child sends exactly one line of text in return. Consequently, `processChild()` just has to read one line of text:

```
    void Monitor::processChild(int fd)
    {
        ifdstreambuf ifdbuf(fd);
        istream istr(&ifdbuf);
        string line;

        getline(istr, line);
        cout << d_child[fd]->pid() << ": " << line << endl;
    }
```

Please note the construction `d_child[fd]->pid()` used in the above source. `Monitor` defines the data member `map<int, Child *> d_child`. This map contains the child's order number as its key, and a pointer to the `Child` object as its value. A pointer is used here, rather than a `Child` object, since we do want to use the facilities offered by the map, but don't want to copy a `Child` object.

The implication of using pointers as `map`-values is of course that the responsibility to destruct the `Child` object once it becomes superfluous now lies with the programmer, and not any more with the run-time support system.

Now that `run()`'s implementation has been covered, we'll concentrate on the various commands users might enter:

- When the `start` command is issued, a new child process is started. A new element is added

to d_child by the member createNewChild(). Next, the Child object should start its activities, but the Monitor object can not wait here for the child process to complete its activities, as there is no well-defined endpoint in the near future, and the user will probably want to enter more commands. Therefore, the Child process will run as a *daemon*: its parent process will terminate immediately, and its own child process will continue in the background. Consequently, createNewChild() calls the child's fork() member. Although it is the child's fork() function that is called, it is still the monitor program wherein fork() is called. So, the *monitor* program is duplicated by fork(). Execution then continues:

- At the Child's parentProcess() in its parent process;
- At the Child's childProcess() in its child process

As the Child's parentProcess() is an empty function, returning immediately, the Child's parent process effectively continues immediately below createNewChild()'s cp->fork() statement. As the child process never returns (see section 23.4.5.3), the code below cp->fork() is never executed by the Child's child process. This is exactly as it should be.

In the parent process, createNewChild()'s remaining code simply adds the file descriptor that's available for reading information from the child to the set of input file descriptors monitored by d_select, and uses d_child to establish the association between that file descriptor and the Child object's address:

```
void Monitor::createNewChild(int, string const &)
{
    Child *cp = new Child(++d_nr);

    cp->fork();

    int fd = cp->readFd();

    d_selector.addReadFd(fd);
    d_child[fd] = cp;

    cerr << "Child " << d_nr << " started\n";
}
```

- Direct communication with the child is required for the stop <nr> and <nr> text commands. The former command terminates child process <nr>, by calling stopChild(). This function locates the child process having the order number using an anonymous object of the class Find, nested inside Monitor. The class Find simply compares the provided nr with the children's order number returned by their nr() members:

```
inline Monitor::Find::Find(int nr)
:
    d_nr(nr)
{}
inline bool Monitor::Find::operator()(
                        std::map<int, Child *>::value_type &vt) const
{
    return d_nr == vt.second->nr();
}
```

If the child process having order number nr was found, its file descriptor is removed from d_selector's set of input file descriptors. Then the child process itself is terminated by the static member killChild(). The member killChild() is declared as a *static* member function, as it is used as function argument of the for_each() generic algorithm by erase() (see below). Here is killChild()'s implementation:

```
void Monitor::killChild(map<int, Child *>::value_type it)
{
```

```
        if (kill(it.second->pid(), SIGTERM))
            cerr << "Couldn't kill process " << it.second->pid() << endl;

        // reap defunct child process
        int status = 0;
        while( waitpid( it.second->pid(), &status, WNOHANG) > -1)
        {};
    }
```

Having terminated the specified child process, the corresponding `Child` object is destroyed and its pointer is removed from `d_child`:

```
    void Monitor::stopChild(int nr, string const &)
    {
        map<int, Child *>::iterator it =
            find_if(d_child.begin(), d_child.end(), Find(nr));

        if (it == d_child.end())
            cerr << "No child number " << nr << endl;
        else
        {
            d_selector.rmReadFd(it->second->readFd());

            delete it->second;
            d_child.erase(it);
        }
    }
```

- The command `<nr> text` will send `text` to child process `nr` using the member function `sendChild()`. This function too, will use a `Find` object to locate the process having order number `nr`, and will then simply insert the text into the writing end of a pipe connected to the indicated child process:

```
    void Monitor::sendChild(int nr, string const &line)
    {
        map<int, Child *>::iterator it =
            find_if(d_child.begin(), d_child.end(), Find(nr));

        if (it == d_child.end())
            cerr << "No child number " << nr << endl;
        else
        {
            ofdnstreambuf ofdn(it->second->writeFd());
            ostream out(&ofdn);

            out << line << endl;
        }
    }
```

- When users enter `exit` the member `exiting()` is called. It terminates all child processes, by visiting all elements of `d_child` using the `for_each()` generic algorithm (see section 19.1.17). The program is subsequently terminated:

```
    void Monitor::exiting(int value, string const &msg)
    {
        for_each(d_child.begin(), d_child.end(), killChild);
        if (msg.length())
            cerr << msg << endl;
        throw value;
    }
```

Finally, the program's `main()` function is simply:

```
#include "monitor.h"

int main()
try
{
    Monitor monitor;

    monitor.run();
}
catch (int exitValue)
{
    return exitValue;
}

/*
    Example of a session:

    # a.out
    ? start
    Child 1 started
    ? 1 hello world
    ? 3394: Child 1:1:  hello world
    ? 1 hi there!
    ? 3394: Child 1:2:  hi there!
    ? start
    Child 2 started
    ? 3394: Child 1: standing by
    ? 3395: Child 2: standing by
    ? 3394: Child 1: standing by
    ? 3395: Child 2: standing by
    ? stop 1
    ? 3395: Child 2: standing by
    ? 2 hello world
    ? 3395: Child 2:1:  hello world
    ? 1 hello world
    No child number 1
    ? exit3395: Child 2: standing by
    ?
    #
*/
```

23.4.5.3 The class 'Child'

When the `Monitor` object starts a child process, it has to create an object of the class `Child`. The `Child` class is derived from the class `Fork`, allowing its construction as a *daemon*, as discussed in the previous section. Since a `Child` object is a daemon, we know that its parent process should be defined as an empty function. its `childProcess()` must of course still be defined. Here are the characteristics of the class `Child`:

- The `Child` class defines two `Pipe` data members, to allow communications between its own child- and parent processes. As these pipes are used by the `Child`'s child process, their names are aimed at the child process: the child process reads from `d_in`, and writes to `d_out`. Here is the interface of the class `Child`:

```
class Child: public Fork
```

```
{
    Pipe                d_in;
    Pipe                d_out;

    int         d_parentReadFd;
    int         d_parentWriteFd;
    int         d_nr;

    public:
        Child(int nr);
        virtual ~Child();
        int readFd() const;
        int writeFd() const;
        int pid() const;
        int nr() const;
        virtual void childRedirections();
        virtual void parentRedirections();
        virtual void childProcess();
        virtual void parentProcess();
};
```

- The `Child`'s constructor simply stores its argument, a child-process order number, in its own `d_nr` data member:

```
inline Child::Child(int nr)
:
    d_nr(nr)
{}
```

- The `Child`'s child process will simply obtain its information from its standard input stream, and it will write its information to its standard output stream. Since the communication channels are pipes, redirections must be configured. The `childRedirections()` member is implemented as follows:

```
void Child::childRedirections()
{
    d_in.readFrom(STDIN_FILENO);
    d_out.writtenBy(STDOUT_FILENO);
}
```

- Although the parent process performs no actions, it must configure some redirections. Since the names of the pipes indicate their functions in the child process, `d_in` is used for *writing* by the parent, and `d_out` is used for *reading* by the parent. Here is the implementation of `parentRedirections()`:

```
void Child::parentRedirections()
{
    d_parentReadFd = d_out.readOnly();
    d_parentWriteFd = d_in.writeOnly();
}
```

- The `Child` object will exist until it is destroyed by the `Monitor`'s `stopChild()` member. By allowing its creator, the `Monitor` object, to access the parent-side ends of the pipes, the `Monitor` object can communicate with the `Child`'s child process via those pipe-ends. The members `readFd()` and `writeFd()` allow the `Monitor` object to access these pipe-ends:

```
inline int Child::readFd() const
{
    return d_parentReadFd;
}
```

```
inline int Child::writeFd() const
{
    return d_parentWriteFd;
}
```

- The `Child` object's child process basically has two tasks to perform:
  - It must reply to information appearing at its standard input stream;
  - If no information has appeared within a certain time frame (the implementations uses an interval of five seconds), then a message should be written to its standard output stream anyway.

  To implement this behavior, `childProcess()` defines a local `Selector` object, adding `STDIN_FILENO` to its set of monitored input file descriptors.

  Then, in an endless loop, `childProcess()` waits for `selector.wait()` to return. When the alarm goes off it sends a message to its standard output. (Hence, into the writing pipe). Otherwise, it will echo the messages appearing at its standard input to its standard output. Here is the implementation of the `childProcess()` member:

```
void Child::childProcess()
{
    Selector    selector;
    size_t    message = 0;

    selector.addReadFd(STDIN_FILENO);
    selector.setAlarm(5);

    while (true)
    {
        try
        {
            if (!selector.wait())       // timeout
                cout << "Child " << d_nr << ": standing by\n";
            else
            {
                string  line;
                getline(cin, line);
                cout << "Child " << d_nr << ":" << ++message << ": " <<
                                                    line << endl;
            }
        }
        catch (...)
        {
                cout << "Child " << d_nr << ":" << ++message << ": " <<
                            "select() failed" << endl;
        }
    }
    exit(0);
}
```

- Next, twoaccessors allow the `Monitor` object to obtain the `Child`'s process ID and order number, respectively:

```
inline int Child::pid() const
{
    return Fork::pid();
}
inline int Child::nr() const
{
    return d_nr;
}
```

- A `Child` process terminates when the user enters a `stop` command. When an existing child process number was entered, the corresponding `Child` object is removed from `Monitor`'s `d_child` map. As a result, its destructor is called. In its turn, `Child`'s destructor will call `kill` to terminate its child, and then waits for the child to terminate. Once the child has terminated, the destructor has completed its work as well and returns, competing the erasure from `d_child`. The implementation offered here will fail if the child process doesn't react to the `SIGTERM` signal. In this demonstration program this does not happen. In 'real life' implementations more elaborate killing-procedures may be required (e.g., using `SIGKILL` in addition to `SIGTERM`). As discussed in section 9.11 it is important to ensure that the destruction succeeds. Here is the implementation of the `Child`'s destructor:

```
Child::~Child()
{
    if (pid())
    {
        cout << "Killing process " << pid() << "\n";
        kill(pid(), SIGTERM);
        int status;
        wait(&status);
    }
}
```

23.5 Function objects performing bitwise operations

In section 18.1 several types of predefined function objects were introduced. Predefined function objects performing arithmetic operations, relational operations, and logical operations exist, corresponding to a multitude of binary- and unary operators.

Some operators appear to be missing: there appear to be no predefined function objects corresponding to bitwise operations. However, their construction is, given the available predefined function objects, not difficult. The following examples show a class template implementing a function object calling the bitwise and (`operator&()`), and a template class implementing a function object calling the unary not (`operator~()`). It is left to the reader to construct similar function objects for other operators.

Here is the implementation of a function object calling the bitwise `operator&()`:

```
#include <functional>

template <typename _Tp>
struct bit_and: public std::binary_function<_Tp, _Tp, _Tp>
{
    _Tp operator()(_Tp const &__x, _Tp const &__y) const
    {
        return __x & __y;
    }
};
```

Here is the implementation of a function object calling `operator~()`:

```
#include <functional>

template <typename _Tp>
struct bit_not: public std::unary_function<_Tp, _Tp>
{
    _Tp operator()(_Tp const &__x) const
    {
        return ~__x;
```

```
        }
    };
```

These and other missing predefined function objects are also implemented in the file `bitfunctional`, which is found in the `cplusplus.yo.zip` archive. It should be noted that these classes are derived from existing class templates (e.g., `std::binary_function` and `std::unary_function`). These base classes offer several typedefs which are expected (used) by various generic algorithms as defined in the STL (cf. chapter 19), thus following the advice offered in, e.g., the **C++** header file `bits/stl_function.h`:

```
*  The standard functors are derived from structs named unary_function
*  and binary_function.  These two classes contain nothing but typedefs,
*  to aid in generic (template) programming.  If you write your own
*  functors, you might consider doing the same.
```

Here is an example using `bit_and()` removing all odd numbers from a vector of `int` values:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include "bitand.h"
using namespace std;

int main()
{
    vector<int> vi;

    for (int idx = 0; idx < 10; ++idx)
        vi.push_back(idx);

    copy
    (
        vi.begin(),
        remove_if(vi.begin(), vi.end(), bind2nd(bit_and<int>(), 1)),
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;
}
/*
    Generated output:

    0 2 4 6 8
*/
```

23.6 Implementing a 'reverse_iterator'

Earlier, in section 21.12.1, the construction of iterators and reverse iteraters was discussed. In that section the iterator was constructed as an inner class in a class derived from a vector of pointers to strings.

An object of this nested iterator class handled the dereferencing of the pointers stored in the vector. This allowed us to sort the *strings* pointed to by the vector's elements rather than the *pointers*.

A drawback of the approach taken in section 21.12.1 is that the class implementing the iterator is closely tied to the derived class as the iterator class was implemented as a nested class. What if we would like to provide any class derived from a container class storing pointers with an iterator handling the pointer-dereferencing?

In this section a variant to the earlier (nested class) approach is discussed. The iterator class will be defined as a *class template*, parameterizing the data type to which the container's elements point as well as the iterator type of the container itself. Once again, we will implement a *RandomIterator* as it is the most complex iterator type.

Our class is named `RandomPtrIterator`, indicating that it is a random iterator operating on pointer values. The class template defines three template type parameters:

- The first parameter specifies the derived class type (`Class`). Like the earlier nested class, `RandomPtrIterator`'s constructor will be private. Therefore we need `friend` declarations to allow client classes to construct `RandomPtrIterators`. However, a `friend class Class` cannot be defined: template parameter types cannot be used in `friend class ...` declarations. But this is no big problem: not every member of the client class needs to construct iterators. In fact, only `Class`'s `begin()` and `end()` members must be able to construct iterators. Using the template's first parameter, friend declarations can be specified for the client's `begin()` and `end()` members.
- The second template parameter parameterizes the container's iterator type (`BaseIterator`);
- The third template parameter indicates the data type to which the pointers point (`Type`).

`RandomPtrIterator` uses one private data element, a `BaseIterator`. Here is the class interface, including the constructor's implementation:

```
#include <iterator>

template <typename Class, typename BaseIterator, typename Type>
class RandomPtrIterator:
      public std::iterator<std::random_access_iterator_tag, Type>
{
    friend RandomPtrIterator<Class, BaseIterator, Type> Class::begin();
    friend RandomPtrIterator<Class, BaseIterator, Type> Class::end();

    BaseIterator d_current;

    RandomPtrIterator(BaseIterator const &current);

    public:
        bool operator!=(RandomPtrIterator const &other) const;
        int operator-(RandomPtrIterator const &rhs) const;
        RandomPtrIterator const operator+(int step) const;
        Type &operator*() const;
        bool operator<(RandomPtrIterator const &other) const;
        RandomPtrIterator &operator--();
        RandomPtrIterator const operator--(int);
        RandomPtrIterator &operator++();
        RandomPtrIterator const operator++(int);
        bool operator==(RandomPtrIterator const &other) const;
        RandomPtrIterator const operator-(int step) const;
        RandomPtrIterator &operator-=(int step);
        RandomPtrIterator &operator+=(int step);
        Type *operator->() const;
};

template <typename Class, typename BaseIterator, typename Type>
RandomPtrIterator<Class, BaseIterator, Type>::RandomPtrIterator(
                                BaseIterator const &current)
:
    d_current(current)
{}
```

Dissecting its `friend` declarations, we see that the members `begin()` and `end()` of a class `Class`, returning a `RandomPtrIterator` object for the types `Class, BaseIterator` and `Type` are granted access to `RandomPtrIterator`'s private constructor. That is exactly what we want. Note that `begin()` and `end()` are declared as *bound friends*.

All `RandomPtrIterator`'s remaining members are public. Since `RandomPtrIterator` is just a generalization of the nested class `iterator` developed in section 21.12.1, re-implementing the required member functions is easy, and only requires us to change `iterator` into `RandomPtrIterator` and to change `std::string` into `Type`. For example, `operator<()`, defined in the class `iterator` as

```
inline bool StringPtr::iterator::operator<(iterator const &other) const
{
    return **d_current < **other.d_current;
}
```

is re-implemented as:

```
    template <typename Class, typename BaseIterator, typename Type>
    bool RandomPtrIterator<Class, BaseIterator, Type>::operator<(
                                    RandomPtrIterator const &other) const
    {
        return **d_current < **other.d_current;
    }
```

As a second example: `operator*()`, defined in the class `iterator` as

```
inline std::string &StringPtr::iterator::operator*() const
{
    return **d_current;
}
```

is re-implemented as:

```
    template <typename Class, typename BaseIterator, typename Type>
    Type &RandomPtrIterator<Class, BaseIterator, Type>::operator*() const
    {
        return **d_current;
    }
```

The pre- and postfix increment operators are re-implemented as:

```
    template <typename Class, typename BaseIterator, typename Type>
    RandomPtrIterator<Class, BaseIterator, Type>
    &RandomPtrIterator<Class, BaseIterator, Type>::operator++()
    {
        ++d_current;
        return *this;
    }
    template <typename Class, typename BaseIterator, typename Type>
    RandomPtrIterator<Class, BaseIterator, Type> const
    RandomPtrIterator<Class, BaseIterator, Type>::operator++(int)
    {
        return RandomPtrIterator(d_current++);
    }
```

Remaining members can be implemented accordingly, their actual implementations are left as an exercise to the reader (or can be obtained from the `cplusplus.yo.zip` archive, of course).

Reimplementing the class `StringPtr` developed in section 21.12.1 is not difficult either. Apart from including the header file defining the class template `RandomPtrIterator`, it requires only a single modification as its `iterator` typedef must now be associated with a `RandomPtrIterator`. Here are the full class interface and inline member definitions:

```
#ifndef INCLUDED_STRINGPTR_H_
#define INCLUDED_STRINGPTR_H_

#include <vector>
#include <string>
#include "iterator.h"

class StringPtr: public std::vector<std::string *>
{
    public:
        typedef RandomPtrIterator
                <
                    StringPtr,
                    std::vector<std::string *>::iterator,
                    std::string
                >
                    iterator;

        typedef std::reverse_iterator<iterator> reverse_iterator;

        iterator begin();
        iterator end();
        reverse_iterator rbegin();
        reverse_iterator rend();
};

inline StringPtr::iterator StringPtr::begin()
{
    return iterator(this->std::vector<std::string *>::begin() );
}
inline StringPtr::iterator StringPtr::end()
{
    return iterator(this->std::vector<std::string *>::end());
}
inline StringPtr::reverse_iterator StringPtr::rbegin()
{
    return reverse_iterator(end());
}
inline StringPtr::reverse_iterator StringPtr::rend()
{
    return reverse_iterator(begin());
}
#endif
```

Including `StringPtr`'s modified header file into the program given in section 21.12.2 will result in a program behaving identically to its earlier version, albeit that `StringPtr::begin()` and `StringPtr::end()` now return iterator objects constructed from a template definition.

23.7 A text to anything converter

The standard **C** library offers conversion functions like `atoi()`, `atol()`, and other functions, which can be used to convert `ASCII-Z` strings to numeric values. In **C++**, these functions are still available, but a

more *type safe* way to convert text to other types is by using objects of the class `std::istringsteam`.

Using the `std::istringstream` class instead of the **C** standard conversion functions may have the advantage of type-safety, but it also appears to be a rather cumbersome alternative. After all, we will have to construct and initialize a `std::istringstream` object first, before we're actually able to extract a value of some type from it. This requires us to use a variable. Then, if the extracted value is actually only needed to initialize some function-parameter, one might wonder whether the additional variable and the `istringstream` construction can somehow be avoided.

In this section we'll develop a class (`A2x`) preventing all the disadvantages of the standard **C** library functions, without requiring the cumbersome definitions of `std::istringstream` objects over and over again. The class is called `A2x` for 'ascii to anything'.

`A2x` objects can be used to obtain a value for any type extractable from `std::istream` objects given its textual representation. Since `A2x` represents the object-variant of the **C** functions, it is not only type-safe but *also* extensible. Consequently, their use is greatly preferred over the standard **C** functions. Here are its characteristics:

- `A2x` is derived from `std::istringstream`, so all members of the class `std::istringstream` are available. Thus, extractions of values of variables can always be performed effortlessly. Here's the class's interface:

```
class A2x: public std::istringstream
{
    public:
        A2x();
        A2x(char const *txt);
        A2x(std::string const &str);

        template <typename Type>
        operator Type();

        template <typename Type>
        Type to();

        A2x &operator=(char const *txt);

        A2x &operator=(std::string const &str);
        A2x &operator=(A2x const &other);
};
```

- `A2x` has a default constructor and a constructor expecting a `std::string` argument. The latter constructor may be used to initialize `A2x` objects with text to be converted (e.g., a line of text obtained from reading a configuration file):

```
inline A2x::A2x()
{}

inline A2x::A2x(char const *txt)                    // initialize from text
:
    std::istringstream(txt)
{}

inline A2x::A2x(std::string const &str)
:
    std::istringstream(str.c_str())
{}
```

- `A2x`'s real strength comes from its `operator Type()` conversion member template. As it is a member template, it will automatically adapt itself to the type of the variable that should be

given a value, obtained by converting the text stored inside the A2x object to the variable's type. When the extraction fails, A2x's inherited good() member will return false.

- However, occasionally, the compiler may not be able to determine which type to convert to. In that case, an *explicit template type* could be used:

```
A2x.operator int<int>();
// or just:
A2x.operator int();
```

As neither syntax looks attractive, the member template to() was provided as well, allowing constructions like:

```
A2x.to<int>();
```

Here is its implementation:

```
template <typename Type>
inline Type A2x::to()
{
    Type t;

    return (*this >> t) ? t : Type();
}
```

allowing for a trivial implementation of operator Type():

```
template <typename Type>
inline A2x::operator Type()
{
    return to<Type>();
}
```

- Once an A2x object is available, it may be reinitialized using its operator=() member:

```
#include "a2x.h"

A2x &A2x::operator=(char const *txt)
{
    clear();        // very important!!! If a conversion failed, the object
                    // remains useless until executing this statement
    str(txt);
    return *this;
}
```

Here are some examples showing its use:

```
int x = A2x("12");          // initialize int x from a string "12"
A2x a2x("12.50");           // explicitly create an A2x object

double d;
d = a2x;                    // assign a variable using an A2x object
cout << d << endl;

a2x = "err";
d = a2x;                    // d is 0: the conversion failed,
cout << d << endl;          // and a2x.good() == false

a2x = " a";                 // reassign a2x to new text
```

```
char c = a2x;               // c now 'a': internally operator>>() is used
cout << c << endl;          // so initial blanks are skipped.

int expectsInt(int x);      // initialize a parameter using an
expectsInt(A2x("1200"));    // anonymous A2x object

d = A2x("12.45").to<int>(); // d is 12, not 12.45
cout << d << endl;
```

A complementary class (`X2a`), converting values to text, can easily be constructed as well. The construction of `X2a` is left as an exercise to the reader.

23.8 Wrappers for STL algorithms

Many generic algorithms (cf. chapter 19) use function objects to operate on the data to which their iterators refer, or they require predicate function objects using some criterion to make a decision about these data. The standard approach followed by the generic algorithms is to pass the information to which the iterators refer to overloaded function call operators (i.e., `operator()()`) of function objects that are passed as arguments to the generic algorithms.

Usually this approach requires the construction of a dedicated class implementing the required function object. However, in many cases the *class context* in which the iterators exist already offers the required functionality. Alternatively, the functionality might exist as member function of the objects to which the iterators refer. For example, finding the first empty `string` object in a vector of `string` objects could profitably use the `string::empty()` member.

Another frequently encountered situation is related to a *local context*. Once again, consider the situation where the elements of a `string` vector are all visited: each object must be inserted in a stream whose reference is only known to the function in which the `string` elements are visited, but some additional information must be passed to the insertion function as well, making the use of the `ostream_inserter` less appropriate.

The frustrating part of using generic algorithms is that these dedicated function objects often very much look like each other, but the standard solution (using predefined function objects using specialized iterators) seldom do the required job: their fixed function interfaces (e.g., `equal_to` calling the object's `operator==()`) often are too rigid to be useful and, furthermore, they are unable to use any additional local context that is active when they are used.

One may wonder whether class templates might be constructed which can be used again and again to create dedicated function objects. Such class template instantiations should offer facilities to call configurable (member) functions using a configurable local context.

In the upcoming sections, several *wrapper templates* supporting these requirements are developed. To support a *local context*, a dedicated *local context struct* is introduced. Furthermore, the wrapper templates will allow us to specify at construction time the member function that should be called. Thus the rigidness of the fixed member function as used in the predefined function objects is avoided.

As an example of a generic algorithm usually requiring a simple function object, consider `for_each()`. The `operator()()` of the function object passed to this algorithm receives as its argument a reference to the object to which the iterators refer. Generally, `operator()()` will do one of two things:

- It may call a member function of the object defined in its parameter list (e.g., `operator()(string &str)` may call `str.length()`);
- It may call a function, passing it its parameter as argument (e.g., calling `somefunction(str)`).

Of course, the latter example is a bit overkill, since `somefunction()`'s address could actually directly have been passed to the generic algorithm, so why use this complex procedure? The answer is *context*: if `somefunction()` would actually require other arguments, representing the local context in

which `somefunction()` was called, then the function object's constructor could have received the local context as its arguments, passing that local context on to `somefunction()`, together with the object received by the function object's `operator()()` function. There is no way to pass any local context to the generic algorithm's simple variant, in which a function's address is passed to the generic function.

At first sight, however, the fact that a local context differs from one situation to another makes it hard to standardize the local context: a local context might consist of values, pointers, references, which differ in number and types from one situation to another. Defining templates for all possible situations is clearly impractical, and using **C**-style variadic functions is also not very attractive, since the arguments passed to a variadic function object constructor cannot simply be passed on to the function object's `operator()()`.

The concept of a *local context struct* is introduced to standardize the local context. It is based on the following considerations:

- Usually, a function requiring a local context is a member function of some class.
- Instead of using the intuitive implementation where the member function is given the required parameters representing a local context, it receives a single argument: a value, pointer or reference to a (possibly `const`) local context.
- The local context is defined in the function's class interface.
- Before the function is called, a local context is initialized, which is then passed as argument to the function.

Of course, the organization of local contexts will differ from one situation to the next situation, but there is always just *one* local context required. The fact that the inner organization of the local context differs from one situation to the next causes no difficulty at all to **C++**'s template mechanism. Actually, having available a generic type (*Context*) together with several concrete instantiations of that generic type is a text-book argument for using templates.

23.8.1 Local context structs

When a function is called, the context in which it is called is made known to the function by providing the function with a parameter list. When the function is called, these parameters are initialized by the function's arguments. For example, a function `show()` may expect two arguments: an `ostream` into which the information is inserted and an object which will be inserted into the stream. For example:

```
void State::show(ostream &out, Item const &item)
{
    out << "Here is item " << item.nr() << ":\n" <<
            item << endl;
}
```

Of course, functions differ greatly in their parameter lists: both the numbers and types of their parameters vary.

A *local context struct* is used to standardize the parameter lists of functions, for the benefit of template construction. In the above example, the function `State::show()` uses a local context consisting of an `ostream &` and an `Item const &`. This context never changes, and may be offered through a `struct` defined as follows:

```
struct ShowContext
{
    ostream &out;
    Item const &item;
};
```

Note how this `struct` mimics `State::show()`'s parameter list. Since it is directly connected to the function `State::show()` it is best defined in the class `State`. Once we have defined this `struct`, `State::show()`'s implementation is modified. It now expects a `ShowContext &`:

```
void State::show(ShowContext &context)
{
    context.out << "Here is item " << context.item.nr() << ":\n" <<
            context.item << endl;
}
```

Using a local context `struct` any parameter list (except those of variadic functions) can be standardized to a parameter list consisting of a single element. Now that we have a single parameter to specify any local context we're ready for the 'templatization' of function object wrapper classes.

23.8.2 Member functions called from function objects

The *function operator* member `operator()()` is characteristic of function objects. It may be defined as a function having various parameters. In the context of generic algorithms, they usually have one or two parameters referring to the data to be processed by the algorithm.

The class template constructor should be aware of the fact that it is not known beforehand whether these parameters are objects, primitive types, pointers or references. Knowing this, however, *is* important when the function object instantiated from the template class is to be used by generic algorithms, since many generic algorithms require that various types are defined by the function object. For example, generic algorithms like `for_each`, calling unary argument function objects, expect that these function objects define the types `argument_type` and `result_type`, referring to the *plain types* of, respectively, the function operator's argument and return value. Analogously, generic algorithms like `includes` (cf. section 19.1.1) expect that function objects define the types `first_argument_type`, `second_argument_type` and `result_type`.

To determine the plain type of a template type parameter a *trait class* can be used. In section section 22.4 the trait class `TypeTrait` was intrioduced, allowing templates to determine what various characteristics are of template type parameters. This `TypeTrait` class can profitably be used here to determine the proper definnitions of types required by generic algorithms.

Let's assume that we would like to create a function object changing all letters in `string` objects into capital letters. Clearly we will have to access the string's individual characters. However, the strings themselves may be made available through references (e.g., when iterating over the elements of a `vector<string>`), but also through pointers (e.g., when iterating over the elements of a `vector<string *>`).

The template class providing the function object should *not* be responsible for the actions to be performed. Rather, executing the required actions should be deferred to a function, which can be specified when the template is instantiated. If that function is a member function it should, for various reasons, be a *static* member function:

- A non-static function needs an object. But will the object be available as a pointer or as a reference to an object? The issue could be solved using trait classes, but pointers and references to objects require different operators (i.e., `.*` and `->*`, respectively), which would add to the complexity of the final template class.
- If a non-static member function is used, will it be a const or non-const member function? Knowing this is important, since the function's address is stored in the function object, and so its prototype must be known.
- A *static* member function leaves the option of calling a member function open, if the context struct contains a pointer or reference to an object for which a non-static member must be called. Since a static member function is a member of its class, a *non-static* member function called for the object specified in the context struct can be a private member function.

Generic algorithms also differ in the way they use the function object's return value. This turns out to be no problem: templates allow us to parameterize the return types of functions.

23.8.3 The unary argument context sensitive Function Object template

As an opening example, let's assume we have a class `Strings` holding a `vector<string> d_vs` data member. We would like to change all occurrences of a specified set of letters found in the strings stored in `d_vs` into uppercase characters, and we would like to insert the original *and* modified strings into a configurable `ostream` object. To accomplish this, our class offers a member `uppercase(ostream &out, char const *letters)`.

We would like to use the `for_each()` generic algorithm. This algorithm may be given a function object. The function object will be initialized with a local context consisting of the `ostream` object and the set of letters to be used. The following support class is constructed:

```
class Support
{
    std::ostream &d_out;
    std::string d_letters;

    public:
        Support(std::ostream &out, char const *letters);
        void operator()(std::string &str) const;
};

inline Support::Support(std::ostream &out, char const *letters)
:
    d_out(out),
    d_letters(letters)
{}

inline void Support::operator()(std::string &str) const
{
    d_out << str << " ";

    for
    (
        std::string::iterator strIter = str.begin();
            strIter != str.end();
                ++strIter
    )
    {
        if (d_letters.find(*strIter) != std::string::npos)
            *strIter = toupper(*strIter);
    }
    d_out << str << std::endl;
}
```

Note that the implementation of `operator()()` contains another `for` statement, which *should* be replaced by another `for_each` call. This suggests that either another function object must be constructed or that an overloaded version of `operator()()` must be defined. Both alternatives are not very attractive: constructing large numbers of small function object classes soon becomes a nuisance and there's a limit imposed by the parameter types to the number of overloaded `operator()()` members that can be defined, let alone that the self-documenting value of the purpose of '`operator()()`' is very limited.

For now an anonymous `Support` class object is used in the implementation of the class `Strings`. Here is an example of its definition and use:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

#include "support.h"

class Strings
{
    std::vector<std::string> d_vs;

    public:
        void uppercase(std::ostream &out, char const *letters);
};

void Strings::uppercase(std::ostream &out, char const *letters)
{
    for_each(d_vs.begin(), d_vs.end(), Support(out, letters));
}

using namespace std;

int main(int argc, char **argv)
{
    Strings s;

    s.uppercase(cout, argv[1]);
}
```

To 'templatize' the `Support` class using the considerations discussed previously, we perform the following steps:

- The local context will be put in a `struct`, which is then passed to the template's constructor, so of the template type parameters should be defined as a reference (or pointer, if that's preferred) to a context struct.
- The implementation of the template's `operator()()` is standardized: it will call a function, receiving the `operator()()`'s argument (which also becomes a template parameter) and a reference to the context as its arguments. The address of the function to call may be stored in a local variable of the function template object. In the `Support` class, `operator()()` uses a `void` return type. This type is usually appropriate, but when defining predicates a `bool` may be required. Therefore, the return type of the template's `operator()()` (and thus the return type of the called function) is made configurable as well, offering a default type `void` for convenience. Thus, we get the following definition of the variable holding the address of the function to call:

  ```
  ReturnType (*d_fun)(Type, Context);
  ```

  and the template's `operator()()`, coining the classname `FnWrap1c` for 'function object wrapper of a unary (1) function object, accepting context information', becomes:

  ```
  template<typename Type, typename Context, typename ReturnType = void>
  inline ReturnType FnWrap1c<Type, Context, ReturnType>::operator()(
                                                  Type param) const
  {
      return (*d_fun)(param, d_context);
  }
  ```

- The template's constructor is given two arguments: a function address and the local context:

  ```
  template <typename Type, typename Context, typename ReturnType>
  ```

```
        FnWrap1c<Type, Context, ReturnType>::FnWrap1c(
                        ReturnType fun(Type, Context),  Context context)
        :
            d_fun(fun),
            d_context(context)
        {}
```

Now we're ready to construct the full class template `FnWrap1c` using the `TypeTraits` class to determine the *argument_type* and `result_type`:

```
#include "typetrait.h"

template <typename Type, typename Context, typename ReturnType = void>
class FnWrap1c
{
    ReturnType (*d_fun)(Type, Context);
    Context d_context;

    public:
        typedef typename TypeTrait<Type>::Plain         argument_type;
        typedef typename TypeTrait<ReturnType>::Plain   result_type;

        FnWrap1c(ReturnType fun(Type, Context), Context context);
        ReturnType operator()(Type param) const;
};

template <typename Type, typename Context, typename ReturnType>
FnWrap1c<Type, Context, ReturnType>::FnWrap1c(
                ReturnType fun(Type, Context), Context context)
:
    d_fun(fun),
    d_context(context)
{}

template <typename Type, typename Context, typename ReturnType>
inline ReturnType FnWrap1c<Type, Context, ReturnType>::operator()(Type param)
                                                                    const
{
    return (*d_fun)(param, d_context);
}
```

Now the template can be used. The class `Support` is abandoned. Instead of using a separate class, all members required to transform the strings of `Strings` objects will be defined as static members inside `Strings` itself. This neatly localizes the actions where they belong: inside the class that wants to transform its own data. So, the original dedicated implementation of `Support::operator()()` is now defined as a static member `xform` inside the class `Strings`. The class also defines a context struct `Context`, containing a reference to the `ostream` to use and a set of characters to capitalize. The public function `uppercase` now simply initializes the context struct, and creates an `FnWrap1c` object, calling `xform`, which is passed to the `for_each` call. Note that `FnWrap1c`'s template type arguments exactly mirror `xform`'s parameter types. This is a general characteristic of the function object wrappers that are introduced in this and the coming section.

It is the purpose of `xform` to transform the characters of an individual string. What better procedure than to call `for_each` again, this time using the string's `begin()` and `end()` members to define another iterator range. In this case the letter set must be known, and so it is passed as the local context to `FnWrap1c` defined inside `xform`. This latter `FnWrap1c` object calls the static function `foundToUpper` to capitalize individual characters of the strings if necessary. Here is the new implementation of the class `Strings`, now using `FnWrap1c`:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

#include "fnwrap1.h"

class Strings
{
    std::vector<std::string> d_vs;

    struct Context
    {
        std::ostream &out;
        std::string letters;
    };

    public:
        void uppercase(std::ostream &out, char const *letters);

    private:
        static void xform(std::string &str, Context &context);
        static void foundToUpper(char &ch, std::string const &letters);
};

void Strings::uppercase(std::ostream &out, char const *letters)
{
    Context context = {out, letters};

    for_each(d_vs.begin(), d_vs.end(),
        FnWrap1c<std::string &, Context &>(xform, context));
}

void Strings::xform(std::string &str, Context &context)
{
    context.out << str << " ";

    for_each(str.begin(), str.end(),
        FnWrap1c<char &, std::string const &>(
                                    foundToUpper, context.letters));

    context.out << str << std::endl;
}

void Strings::foundToUpper(char &ch, std::string const &letters)
{
    if (letters.find(ch) != std::string::npos)
        ch = toupper(ch);
}

using namespace std;

int main(int argc, char **argv)
{
    Strings s;

    s.uppercase(cout, argv[1]);
}
```

To illustrate the use of the ReturnType template parameter, let's assume that the transformations are only required up to the first empty string. In this case, the find_if generic algorithm comes in handy, since it stops once a predicate returns true. In that case the xform() function should return a bool value, and the uppercase() implementation specifies an explicit type (bool) for the ReturnType template parameter. The next implementation of the class Strings merely shows the required modifications, the remainder is not altered:

```
class Strings
{
    private:
        static bool xform(std::string &str, Context &context);
};

void Strings::uppercase(std::ostream &out, char const *letters)
{
    Context context = {out, letters};

    find_if(d_vs.begin(), d_vs.end(),
        FnWrap1c<std::string &, Context &, bool>(xform, context));
}

bool Strings::xform(std::string &str, Context &context)
{
    if (str.empty())
        return true;

    // previous implementation (not modified)

    return false;
}
```

23.8.4 The binary argument context sensitive Function Object template

Having constructed the unary template wrapper, the construction of the binary template wrapper should offer no surprises. The function object's operator()() is now called with two, rather than one argument. Coining the classname FnWrap2c, it's implementation is almost identical to FnWrap1c's implementation, and it can be found in the Bobcat library[3]. . An example showing its use is:

```
// accumulating strings from a vector to one big string, using
// 'accumulate'.
#include <iostream>
#include <numeric>
#include <string>
#include <vector>
#include <bobcat/fnwrap2c>

using namespace std;
using namespace FBB;

class Strings
{
    vector<string> d_vs;

    public:
        Strings()
        {
```

[3] http://bobcat.sourceforge.net

```
            d_vs.push_back("one");
            d_vs.push_back("two");
            d_vs.push_back("three");
        }

        void display(ostream &out) const
        {
            SContext c = {1, out};

            cout << "On Exit: " <<
                accumulate(
                    d_vs.begin(), d_vs.end(),
                    string("HI"),
                    FnWrap2c<string const &, string const &,
                             SContext &, string>(&show, c)
                ) <<
                endl;
        }

    private:
        struct SContext
        {
            size_t nr;
            ostream &out;
        };

        static string show(string const &str1,
                           string const &str2,
                           SContext &c)
        {
            c.out << c.nr++ << " " << str1 << " " << str2 <<
                     endl;
            return str1 + " " + str2;
        }
};

int main()
{
    Strings s;
    s.display(cout);
}
```

As with the unary template wrapper (see section 23.8.3), an additional class may be defined that does not require a local context. After compilation and linking, just run the program without any arguments. It requires Bobcat's `fnwrap2c` and `typetrait` header files.

23.9 Using 'bisonc++' and 'flex'

The example discussed in this section digs into the peculiarities of using a parser- and scanner generator generating **C++** sources. Once the input for a program exceeds a certain level of complexity, it's advantageous to use a scanner- and parser-generator to create the code which does the actual input recognition.

The current example assumes that the reader knows how to use the scanner generator `flex` and the parser generator `bison`. Both `bison` and `flex` are well documented elsewhere. The original predecessors of `bison` and `flex`, called `yacc` and `lex` are described in several books, e.g. in O'Reilly's book

'lex & yacc'[4].

However, scanner- and parser generators are also (and maybe even more commonly, nowadays) available as free software. Both bison and flex are usually part of software distributions or they can be obtained from ftp://prep.ai.mit.edu/pub/non-gnu. Flex creates a C++ class when %option c++ is specified.

For parser generators the program bison is available. Back in the early 90's *Alain Coetmeur* (coetmeur@icdc.fr[5]) created a **C++** variant (bison++) creating a parser class. Although bison++ program produces code that can be used in **C++** programs it also shows many characteristics that are more appropriate in a **C** context than in a **C++** context. In January 2005 I rewrote parts of Alain's bison++ program, resulting in the original version of the program **bisonc++**. Then, in May 2005 a complete rewrite of the bisonc++ parser generator was completed, which is available on the Internet having version numbers 0.98 and beyond. Bisonc++ can be downloaded from http://bisoncpp.sourceforge.net/, where it is available as source archive and as binary (i386) Debian[6] binary package (including bisonc++'s documentation). Bisonc++ creates a cleaner parser class setup than bison++. In particular, it derives the parser class from a base-class, containing the parser's token- and type-definitions as well as all member functions which should not be (re)defined by the programmer. Most of these members might also be defined directly in the parser class. Because of this approach, the resulting parser class is very small, declaring only members that are actually defined by the programmer (as well as some other members, generated by bisonc++ itself, implementing the parser's parse() member). Actually, parse() is initially the *only* public member of bisonc++'s generated parser class. Remaining members are private. The only member which is *not* implemented by default is lex(), producing the next lexical token. When the directive %scanner (see section 23.9.2.1) is used, bisonc++ will generate a standard implementation for this member; otherwise it must be implemented by the programmer.

In this section of the Annotations we will focus on bisonc++ as our *parser generator*.

Using flex and bisonc++ class-based scanners and parsers can be generated. The advantage of this approach is that the interface to the scanner and the parser tends to become cleaner than without using the class interface. Furthermore, classes allow us to get rid of most if not all global variables, making it easy to use multiple parsers in one program.

Below two examples are elaborated. The first example only uses flex. The scanner it generates monitors the production of a file from several parts. This example focuses on the lexical scanner, and on switching files while churning through the information. The second example uses both flex and bisonc++ to generate a scanner and a parser transforming standard arithmetic expressions to their postfix notations, commonly used in code generated by compilers and in HP-calculators. In the second example the emphasis is mainly on bisonc++ and on composing a scanner object inside a generated parser.

23.9.1 Using 'flex' to create a scanner

The lexical scanner developed in this section is used to monitor the production of a file from several subfiles. The setup is as follows: the input-language knows of an #include directive, followed by a text string specifying the file (path) which should be included at the location of the #include.

In order to avoid complexities irrelevant to the current example, the format of the #include statement is restricted to the form #include <filepath>. The file specified between the pointed brackets should be available at the location indicated by filepath. If the file is not available, the program terminates after issuing an error message.

The program is started with one or two filename arguments. If the program is started with just one filename argument, the output is written to the standard output stream cout. Otherwise, the output is written to the stream whose name is given as the program's second argument.

The program defines a maximum nesting depth. Once this maximum is exceeded, the program termi-

[4] http://www.oreilly.com/catalog/lex
[5] mailto:coetmeur@icdc.fr
[6] http://www.debian.org

nates after issuing an error message. In that case, the filename stack indicating where which file was included is printed.

One additional feature is that (standard **C++**) comment-lines are ignored. So, `include` directives in comment-lines are ignored too.

The program is created along the following steps:

- First, the file `lexer` is constructed, containing the input-language specifications.
- From the specifications in `lexer` the requirements for the `class Scanner` evolve. The `Scanner` class is a wrapper around the class `yyFlexLexer` generated by `flex`. The requirements result in the interface specification for the class `Scanner`.
- Next, `main()` is constructed. A `Scanner` object is created inspecting the command-line arguments. If successful, the scanner's member `yylex()` is called to construct the output file.
- Now that the global setup of the program has been specified, the member functions of the various classes are constructed.
- Finally, the program is compiled and linked.

23.9.1.1 The derived class 'Scanner'

The code associated with the regular expression rules is located inside the class `yyFlexLexer`. However, we would of course want to use the derived class's members in this code. This causes a little problem: how does a base-class member know about members of classes derived from it?

Fortunately, inheritance helps us to implement this. In the specification of the class `yyFlexLexer()`, we notice that the function `yylex()` is a *virtual* function. The header file `FlexLexer.h` declares the `virtual` member `int yylex()`:

```
class yyFlexLexer: public FlexLexer
{
    public:
        yyFlexLexer( istream* arg_yyin = 0, ostream* arg_yyout = 0 );

        virtual ~yyFlexLexer();

        void yy_switch_to_buffer( struct yy_buffer_state* new_buffer );
        struct yy_buffer_state* yy_create_buffer( istream* s, int size );
        void yy_delete_buffer( struct yy_buffer_state* b );
        void yyrestart( istream* s );

        virtual int yylex();

        virtual void switch_streams( istream* new_in, ostream* new_out );
};
```

As this function is a virtual function it can be overridden in a *derived* class. In that case the overridden function will be called from its base class (i.e., `yyFlexLexer`) code. Since the derived class's `yylex()` is called, it will now have access to the members of the derived class, and also to the public and protected members of its base class.

By default, the context in which the generated scanner is placed is the function `yyFlexLexer::yylex()`. This context changes if we use a derived class, e.g., `Scanner`. To derive `Scanner` from `yyFlexLexer`, generated by `flex`, do as follows:

- The function `yylex()` must be declared in the derived class `Scanner`.

- *Options* (see below) are used to inform `flex` about the derived class's name.

Looking at the regular expressions themselves, notice that we need rules to recognize comment, `#include` directives, and all remaining characters. This is all fairly standard practice. When an `#include` directive is detected, the directive is parsed by the scanner. This too is common practice. Here is what our lexical scanner will do:

- As usual, preprocessor directives are not analyzed by a parser, but by the lexical scanner;
- The scanner uses a mini scanner to extract the filename from the directive, throwing a `Scanner::Error` value (`invalidInclude`) if this fails;
- If the filename could be extracted, it is stored in `nextSource`;
- When the `#include` directive has been processed, `pushSource()` is called to perform the switch to another file;
- When the end of the file (`EOF`) is reached, the derived class's member function `popSource()` is called, popping the previously pushed file and returning `true`;
- Once the file-stack is empty, `popSource()` returns `false`, resulting in calling `yyterminate()`, terminating the scanner.

The lexical scanner specification file is organized similarly as the one used for `flex` in **C** contexts. However, for **C++** contexts, `flex` may create a class (`yyFlexLexer`) from which another class (e.g., `Scanner`) can be derived. The flex specification file itself has three sections:

- The lexer specification file's first section is a **C++** *preamble*, containing code which can be used in the code defining the actions to be performed once a regular expression is matched. In the current setup, where each class has its own *internal header file*, the internal header file includes the file `scanner.h`, in turn including `FlexLexer.h`, which is part of the `flex` distribution.

  However, due to the complex setup of this latter file, it should not be read again by the code generated by `flex`. So, we now have the following situation:

  - First we look at the lexer specification file. It contains a preamble including `scanner.ih`, since this declares, via `scanner.h` the class `Scanner`, so that we're able to call `Scanner`'s members from the code associated with the regular expressions defined in the lexer specification file.
  - In `scanner.h`, defining `class Scanner`, the header file `FlexLexer.h`, declaring `Scanner`'s base class, *must* have been read by the compiler before the class `Scanner` itself is defined.
  - Code generated by `flex` already includes `FlexLexer.h`, and as mentioned, `FlexLexer.h` may not be read again. However, `flex` will also insert the specification file's preamble into the code it generates.
  - Since this preamble includes `scanner.ih`, and so `scanner.h`, and so `FlexLexer.h`, we now *do* include `FlexLexer.h` twice in code generated by `flex`. This must be prevented.

  To prevent multiple inclusions of `FlexLexer.h` the following is suggested:

  - Although `scanner.ih` includes `scanner.h`, `scanner.h` itself is modified such that it includes `FlexLexer.h`, *unless* the **C** preprocesser variable `_SKIP_FLEXLEXER_` is defined.
  - In `flex`' specification file `_SKIP_FLEXLEXER_` is defined just prior to including `scanner.ih`.

  Using this scheme, code generated by `flex` will now re-include `FlexLexer.h`. At the same time, compiling `Scanner`'s members proceeds independently of the lexer specification file's preamble, so here `FlexLexer.h` is properly included too. Here is the specification files' preamble:

```
%{
    #define _SKIP_YYFLEXLEXER_
    #include "scanner.ih"
%}
```

- The specification file's second section is a `flex` *symbol area*, used to define symbols, like a mini scanner, or *options*. The following options are suggested:
  - `%option 8bit`: this allows the generated lexical scanner to read 8-bit characters (rather than 7-bit, which is the default).
  - `%option c++`: this results in `flex` generating **C++** code.
  - `%option debug`: this will include *debugging* code into the code generated by `flex`. Calling the member function `set_debug(true)` will activate this debugging code run-time. When activated, information about which rules are matched is written to the standard error stream. To suppress the execution of debug code the member function `set_debug(false)` may be called.
  - `%option noyywrap`: when the scanner reaches the end of file, it will (by default) call a function `yywrap()` which may perform the switch to another file to be processed. Since there exist alternatives which render this function superfluous (see below), it is suggested to specify this option as well.
  - `%option outfile="yylex.cc"`: this defines `yylex.cc` as the name of the generated **C++** source file.
  - `%option warn`: this option is strongly suggested by the `flex` documentation, so it's mentioned here as well. See `flex`' documentation for details.
  - `%option yyclass="Scanner"`: this defines `Scanner` as the name of the class derived from `yyFlexLexer`.
  - `%option yylineno`: this option causes the lexical scanner to keep track of the line numbers of the files it is scanning. When processing nested files, the variable `yylineno` is not automatically reset to the last line number of a file, when returning to a partially processed file. In those cases, `yylineno` will explicitly have to be reset to a former value. If specified, the current line number is returned by the public member `lineno()`, returning an `int`.

  Here is the specification files' symbol area:

```
%option yyclass="Scanner" outfile="yylex.cc" c++ 8bit warn noyywrap yylineno
%option debug

%x      comment
%x      include

eolnComment     "//".*
anyChar         .|\n
```

- The specification file's third section is a *rules section*, in which the regular expressions and their associated actions are defined. In the example developed here, the lexer should copy information from the `istream *yyin` to the `ostream *yyout`. For this the predefined macro `ECHO` can be used. Here is the specification files' symbol area:

```
%%
    /*
        The comment-rules: comment lines are ignored.
    */
{eolnComment}
"/*"                    BEGIN comment;
<comment>{anyChar}
<comment>"*/"           BEGIN INITIAL;

    /*
        File switching: #include <filepath>
    */
#include[ \t]+"<"       BEGIN include;
<include>[^ \t>]+       d_nextSource = yytext;
<include>">"[ \t]*\n    {
```

```
                            BEGIN INITIAL;
                            pushSource(YY_CURRENT_BUFFER, YY_BUF_SIZE);
                        }
<include>{anyChar}      throw invalidInclude;

    /*
        The default rules: eating all the rest, echoing it to output
    */
{anyChar}               ECHO;

    /*
        The <<EOF>> rule: pop a pushed file, or terminate the lexer
    */
<<EOF>>                 {
                            if (!popSource(YY_CURRENT_BUFFER))
                                yyterminate();
                        }
%%
```

Since the derived class's members may now access the information stored within the lexical scanner itself (it can even access the information *directly*, since the data members of `yyFlexLexer` are protected, and thus accessible to derived classes), most processing can be left to the derived class's member functions. This results in a very clean setup of the lexer specification file, requiring no or hardly any code in the *preamble*.

23.9.1.2 Implementing 'Scanner'

The `class Scanner`, derived as usual from the class `yyFlexLexer`, is generated by **flex**(1). The derived class has access to data controlled by the lexical scanner. In particular, it has access to the following data members:

- `char *yytext`, containing the text matched by a regular expression. Clients may access this information using the scanner's `YYText()` member;
- `int yyleng`, the length of the text in `yytext`. Clients may access this value using the scanner's `YYLeng()` member;
- `int yylineno`: the current line number. This variable is only maintained if `%option yylineno` is specified. Clients may access this value using the scanner's `lineno()` member.

Other members are available as well, but are used less often. Details can be found in `FlexLexer.h`.

Objects of the `class Scanner` perform two tasks:

- They push file information about the current file to a file stack;
- They pop the last-pushed information from the stack once `EOF` is detected in a file.

Several member functions are used to accomplish these tasks. As they are auxiliary to the scanner, they are private members. In practice, develop these private members once the need for them arises. Note that, apart from the private member functions, several private data members are defined as well. Let's have a closer look at the implementation of the class `Scanner`:

- First, we have a look at the class's initial section, showing the conditional inclusion of `FlexLexer.h`, its `class` opening, and its private data. At the top of the class interface the private struct `FileInfo` is defined. `FileInfo` is used to store the names and pointers to open files. The struct has two constructors: one merely accepting a filename, the other also expecting a `bool` argument

indicating that the file is already open and should not be handled by `FileInfo`. This former constructor is used only once: as the initial stream is an already open file there is no need to open it again and so `Scanner`'s constructor will use this constructor to store the name of the initial file only. `Scanner`'s public section starts off by defining the `enum Error` defining various symbolic constants for errors that may be detected:

```
#if ! defined(_SKIP_YYFLEXLEXER_)
#include <FlexLexer.h>
#endif

class Scanner: public yyFlexLexer
{
    struct FileInfo
    {
        std::string d_name;
        std::ifstream    *d_in;

        FileInfo(std::string name)
        :
            d_name(name),
            d_in(new std::ifstream(name.c_str()))
        {}
        FileInfo(std::string name, bool)
        :
            d_name(name),
            d_in(0)
        {}
//        inline bool operator==(FileInfo const &rhs) const
//        {
//            return d_name == rhs.d_name;
//        }
    };

    friend bool operator==(FileInfo const &fi, std::string const &name);

    std::stack<yy_buffer_state *>   d_state;
    std::vector<FileInfo>           d_fileInfo;
    std::string                     d_nextSource;

    static size_t const           s_maxDepth = 10;

    public:
        enum Error
        {
            invalidInclude,
            circularInclusion,
            nestingTooDeep,
            cantRead,
        };
```

- As they are objects, the class's data members are initialized automatically by `Scanner`'s constructor. It activates the initial input (and output) file and pushes the name of the initial input file, using the second `FileInfo` constructor. Here is its implementation:

```
    #include "scanner.ih"

    Scanner::Scanner(istream *yyin, string const &initialName)
    {
        switch_streams(yyin, yyout);
        d_fileInfo.push_back(FileInfo(initialName, false));
```

```
}
```

- The scanning process proceeds as follows: once the scanner extracts a filename from an `#include` directive, a switch to another file is performed by `pushSource()`. If the filename could not be extracted, the scanner throws an `invalidInclude` exception value. The `pushSource()` member and the matching function `popSource()` handle file switching. Switching to another file proceeds as follows:
  - First, the current depth of the `include`-nesting is inspected. If `s_maxDepth` is reached, the stack is considered full, and the scanner throws a `nestingTooDeep` exception.
  - Next, `throwOnCircularInclusion()` is called to avoid circular inclusions when switching to new files. This function throws an exception if a filename is included twice using a simple literal name check. Here is its implementation:

```
#include "scanner.ih"

inline bool operator==(Scanner::FileInfo const &fi, string const &name)
{
    return fi.d_name == name;
}

void Scanner::throwOnCircularInclusion()
{
    vector<FileInfo>::iterator
        it = find(d_fileInfo.begin(), d_fileInfo.end(), d_nextSource);

    if (it != d_fileInfo.end())
        throw circularInclusion;
}
```

  - Then the new filename is added to the `FileInfo` vector, at the same time creating a new `ifstream` object. If this fails, the scanner throws a `cantRead` exception.
  - Finally, a new `yy_buffer_state` is created for the newly opened stream, and the lexical scanner is instructed to switch to that stream using `yyFlexLexer`'s member function `yy_switch_to_buffer()`.

  Here is `pushSource()`'s implementation:

```
#include "scanner.ih"

void Scanner::pushSource(yy_buffer_state *current, size_t size)
{
    if (d_state.size() == s_maxDepth)
        throw nestingTooDeep;

    throwOnCircularInclusion();
    d_fileInfo.push_back(FileInfo(d_nextSource));

    ifstream *newStream = d_fileInfo.back().d_in;

    if (!*newStream)
        throw cantRead;

    d_state.push(current);
    yy_switch_to_buffer(yy_create_buffer(newStream, size));
}
```

- The class `yyFlexLexer` provides a series of member functions that can be used to switch files. The file-switching capability of a `yyFlexLexer` object is founded on the `struct yy_buffer_state`, containing the state of the *scan-buffer* of the currently read file. This buffer is pushed on the

d_state stack when an #include is encountered. Then yy_buffer_state's contents are replaced by the buffer created for the file to be processed next. Note that in the flex specification file the function pushSource() is called as

```
pushSource(YY_CURRENT_BUFFER, YY_BUF_SIZE);
```

YY_CURRENT_BUFFER and YY_BUF_SIZE are macros that are *only* available in the rules section of the lexer specification file, so they must be passed as arguments to pushSource(). Currently it is *not* possible to use these macros in the Scanner class's member functions directly.

- Note that yylineno is not updated when a file switch is performed. If line numbers are to be monitored, then the current value of yylineno should be pushed on a stack, and yylineno should be reset by pushSource(), whereas popSource() should reinstate a former value of yylineno by popping a previously pushed value from the stack. Scanner's current implementation maintains a simple stack of yy_buffer_state pointers. Changing that into a stack of pair<yy_buffer_state *, size_t> elements would allow us to save (and restore) line numbers as well. This modification is left as an exercise to the reader.

- The member function popSource() is called to pop the previously pushed buffer from the stack, allowing the scanner to continue its scan just beyond the just processed #include directive. The member popSource() first inspects the size of the d_state stack: if empty, false is returned and the function terminates. If not empty, then the current buffer is deleted, to be replaced by the state waiting on top of the stack. The file switch is performed by the yyFlexLexer members yy_delete_buffer() and yy_switch_to_buffer(). Note that yy_delete_buffer() does *not* close the ifstream does *not* delete the memory allocated for this stream by pushSource(). Therefore the delete is called for the ifstream pointer stored at the back of d_fileInfo to take care of both. Following this the last FileInfo entry is removed from d_fileInfo. Finally the function returns true:

```
#include "scanner.ih"

bool Scanner::popSource(yy_buffer_state *current)
{
    if (d_state.empty())
        return false;

    yy_delete_buffer(current);
    yy_switch_to_buffer(d_state.top());
    d_state.pop();

    delete d_fileInfo.back().d_in;      // closes the stream as well
    d_fileInfo.pop_back();

    return true;
}
```

- Two service members are offered: stackTrace() dumps the names of the currently pushed files to the standard error stream. It may be called by exception catchers. Here is its implementation:

```
#include "scanner.ih"

void Scanner::stackTrace()
{
    for (size_t idx = 0; idx < d_fileInfo.size() - 1; ++idx)
        cerr << idx << ": " << d_fileInfo[idx].d_name  << " included " <<
                               d_fileInfo[idx + 1].d_name  << endl;
}
```

- lastFile() returns the name of the currently processed file. It may be implemented inline:

```
inline std::string const &Scanner::lastFile()
```

```
{
    return d_fileInfo.back().d_name;
}
```

- The lexical scanner itself is defined in `Scanner::yylex()`. Therefore, `int yylex()` must be declared by the class `Scanner`, as it overrides `FlexLexer`'s virtual member `yylex()`.

23.9.1.3 Using a 'Scanner' object

The program using our `Scanner` is very simple. It expects a filename indicating where to start the scanning process. Initially the number of arguments is checked. If at least one argument was given, then an `ifstream` object is created. If this object can be created, then a `Scanner` object is constructed, receiving the address of the `ifstream` object and the name of the initial input file as its arguments. Then the `Scanner` object's `yylex()` member is called. The scanner object throws `Scanner::Error` exceptions if it fails to perform its tasks properly. These exceptions are caught near `main()`'s end. Here is the program's source:

```
#include "lexer.h"
using namespace std;

int main(int argc, char **argv)
{
    if (argc == 1)
    {
        cerr << "Filename argument required\n";
        return 1;
    }

    ifstream yyin(argv[1]);
    if (!yyin)
    {
        cerr << "Can't read " << argv[1] << endl;
        return 1;
    }

    Scanner scanner(&yyin, argv[1]);
    try
    {
        return scanner.yylex();
    }
    catch (Scanner::Error err)
    {
        char const *msg[] =
        {
            "Include specification",
            "Circular Include",
            "Nesting",
            "Read",
        };
        cerr << msg[err] << " error in " << scanner.lastFile() <<
                            ", line " << scanner.lineno() << endl;
        scanner.stackTrace();
        return 1;
    }
    return 0;
}
```

23.9.1.4 Building the program

The final program is constructed in two steps. These steps are given for a Unix system, on which `flex` and the Gnu **C++** compiler `g++` have been installed:

- First, the lexical scanner's source is created using `flex`. For this the following command can be given:

```
flex lexer
```

- Next, all sources are compiled and linked. In situations where the default `yywrap()` function is used, the `libfl.a` library should be linked against the final program. Normally, that's not required, and the program can be constructed as, e.g.:

```
g++ -o lexer *.cc
```

For the purpose of debugging a lexical scanner, the matched rules and the returned tokens provide useful information. When the `%option debug` was specified, debugging code will be included in the generated scanner. To obtain debugging info, this code must also be activated. Assuming the scanner object is called `scanner`, the statement

```
scanner.set_debug(true);
```

will produce debugging info to the standard error stream.

23.9.2 Using both 'bisonc++' and 'flex'

When an input language exceeds a certain level of complexity, a *parser* is often used to control the complexity of the input language. In this case, a *parser generator* can be used to generate the code verifying the input's grammatical correctness. The lexical scanner (preferably composed into the parser) provides chunks of the input, called *tokens*. The parser then processes the series of tokens generated by its lexical scanner.

Starting point when developing programs that use both parsers and scanners is the grammar. The grammar defines a *set of tokens* which can be returned by the lexical scanner (commonly called the *lexer*).

Finally, auxiliary code is provided to 'fill in the blanks': the actions performed by the parser and by the lexer are not normally specified literally in the grammatical rules or lexical regular expressions, but should be implemented in *member functions*, called from within the parser's rules or which are associated with the lexer's regular expressions.

In the previous section we've seen an example of a **C++** class generated by `flex`. In the current section we concentrate on the parser. The parser can be generated from a grammar specification, processed by the program `bisonc++`. The grammar specification required for `bisonc++` is similar to the specifications required for `bison` (and an existing program `bison++`, written in the early nineties by the Frenchman *Alain Coetmeur*), but `bisonc++` generates a **C++** which more closely follows present-day standards than `bison++`, which still shows many **C**-like features.

In this section a program is developed converting *infix expressions*, in which binary operators are written between their operands, to *postfix expressions*, in which binary operators are written behind their operands. Furthermore, the unary operator - will be converted from its prefix notation to a postfix form. The unary + operator is ignored as it requires no further actions. In essence our little calculator is a micro compiler, transforming numeric expressions into assembly-like instructions.

Our calculator will recognize a very basic set of operators: multiplication, addition, parentheses, and the unary minus. We'll distinguish real numbers from integers, to illustrate a subtlety in bison-like grammar specifications. That's all. The purpose of this section is, after all, to illustrate the construction

of a **C++** program that uses both a parser and a lexical scanner, rather than to construct a full-fledged calculator.

In the coming sections we'll develop the grammar specification for `bisonc++`. Then, the regular expressions for the scanner are specified according to `flex`' requirements. Finally the program is constructed.

23.9.2.1 The 'bisonc++' specification file

The grammar specification file required by `bisonc++` is comparable to the specification file required by `bison`. Differences are related to the class nature of the resulting parser. Our calculator will distinguish real numbers from integers, and will support a basic set of arithmetic operators.

`Bisonc++` should be used as follows:

- As usual, a grammar must be defined. With `bisonc++` this is no different, and `bisonc++` grammar definitions are for all practical purposes identical to `bison`'s grammar definitions.
- Having specified the grammar and (usually) some declarations `bisonc++` is able to generate files defining the parser class and the implementation of the member function `parse()`.
- All class members (except those that are required for the proper functioning of the member `parse()`) must be implemented by the programmer. Of course, they should also be declared in the parser class's header. At the very least the member `lex()` must be implemented. This member is called by `parse()` to obtain the next available token. However, `bisonc++` offers a facility providing a standard implementation of the function **lex()**. The member function `error(char const *msg)` is given a simple default implementation which may be modified by the programmer. The member function `error()` is called when `parse()` detects (syntactic) errors.
- The parser can now be used in a program. A very simple example would be:

```
int main()
{
    Parser parser;
    return parser.parse();
}
```

The `bisonc++` specification file consists of two sections:

- The *declaration section*. In this section bison's tokens, and the priority rules for the operators are declared. However, `bisonc++` also supports several new declarations. These new declarations are important and are discussed below.
- The *rules section*. The grammatical rules define the grammar. This section is identical to the one required by `bison`, albeit that some members that were available in `bison` and `bison++` are considered obsolete in `bisonc++`, while other members can now be used in a wider context. For example, **ACCEPT()** and **ABORT()** can be called from any member called from the parser's action blocks to terminate the parsing process.

Readers familiar with `bison` should note that there is no *header section* anymore. Header sections are used by bison to provide for the necessary declarations allowing the compiler to compile the **C** function generated by `bison`. In **C++** declarations are part of or already used by class definitions. Therefore, a parser generator generating a **C++** class and some of its member functions does not require a header section anymore.

The declaration section The declaration section contains several declarations, among which all tokens used in the grammar and the priority rules of the mathematical operators. Moreover, several new and important specifications can be used here. Those that are relevant to our current example and

only available in bisonc++ are discussed here. The readery is referred to bisonc++'s *man-page* for a full description.

- **%baseclass-header** header
  Defines the pathname of the file to contain (or containing) the parser's base class. Defaults to the name of the parser class plus the suffix base.h.

- **%baseclass-preinclude** header
  Use header as the pathname to the file pre-included in the parser's base-class header. This declaration is useful in situations where the base class header file refers to types which might not yet be known. E.g., with %union a std::string * field might be used. Since the class std::string might not yet be known to the compiler once it processes the base class header file we need a way to inform the compiler about these classes and types. The suggested procedure is to use a pre-include header file declaring the required types. By default header will be surrounded by double quotes (using, e.g., #include "header"). When the argument is surrounded by angle brackets #include <header> will be included. In the latter case, quotes might be required to escape interpretation by the shell (e.g., using -H '<header>').

- **%class-header** header
  Defines the pathname of the file to contain (or containing) the parser class. Defaults to the name of the parser class plus the suffix .h

- **%class-name** parser-class-name
  Declares the class name of this parser. This declaration replaces the %name declaration previously used by bison++. It defines the name of the **C++** class that will be generated. Contrary to **bison++**'s %name declaration, %class-name may appear anywhere in the first section of the grammar specification file. It may be defined only once. If no %class-name is specified the default class name Parser will be used.

- **%debug**
  Provides parse() and its support functions with debugging code, showing the actual parsing process on the standard output stream. When included, the debugging output is active by default, but its activity may be controlled using the setDebug(bool on-off) member. Note that no #ifdef DEBUG macros are used anymore. By rerunning bisonc++ without the -debug option an equivalent parser is generated not containing the debugging code.

- **%filenames** header
  Defines the generic name of all generated files, unless overridden by specific names. By default the generated files use the class-name as the generic file name.

- **%implementation-header** header
  Defines the pathname of the file to contain (or containing) the implementation header. Defaults to the name of the generated parser class plus the suffix .ih. The implementation header should contain all directives and declarations *only* used by the implementations of the parser's member functions. It is the only header file that is included by the source file containing parse()'s implementation. It is suggested that user defined implementations of other class members use the same convention, thus concentrating all directives and declarations that are required for the compilation of other source files belonging to the parser class in one header file.

- **%parsefun-source** source
  Defines the pathname of the file containing the parser member parse(). Defaults to parse.cc.

- **%scanner** header
  Use header as the pathname to the file pre-included in the parser's class header. This file should define a class Scanner, offering a member int yylex() producing the next token from the input stream to be analyzed by the parser generated by bisonc++. When this option is used the parser's member int lex() will be predefined as (assuming the parser class name is Parser):

```
inline int Parser::lex()
{
    return d_scanner.yylex();
}
```

and an object `Scanner d_scanner` will be composed into the parser. The `d_scanner` object will be constructed using its default constructor. If another constructor is required, the parser class may be provided with an appropriate (overloaded) parser constructor after having constructed the default parser class header file using `bisonc++`. By default `header` will be surrounded by double quotes (using, e.g., `#include "header"`). When the argument is surrounded by angle brackets `#include <header>` will be included.

- **%stype typename**
  The type of the semantic value of tokens. The specification `typename` should be the name of an unstructured type (e.g., `size_t`). By default it is `int`. See `YYSTYPE` in `bison`. It should not be used if a `%union` specification is used. Within the parser class, this type may be used as `STYPE`.

- **%union** `union-definition`
  Acts identically to the `bison` declaration. As with `bison` this generates a union for the parser's semantic type. The union type is named `STYPE`. If no `%union` is declared, a simple stack-type may be defined using the `%stype` declaration. If no `%stype` declaration is used, the default stacktype (`int`) is used.

An example of a `%union` declaration is:

```
%union
{
    int     i;
    double  d;
};
```

A union cannot contain objects as its fields, as constructors cannot be called when a union is created. This means that a `string` cannot be a member of the union. A `string *`, however, *is* a possible union member. By the way: the lexical scanner does not have to know about such a union. The scanner can simply pass its scanned text to the parser through its `YYText()` member function. For example using a statement like

```
$$.i = A2x(scanner.YYText());
```

matched text may be converted to a value of an appropriate type.

Tokens and non-terminals can be associated with union fields. This is strongly advised, as it prevents type mismatches, since the compiler will be able to check for type correctness. At the same time, the bison specific variabels `$$`, `$1`, `$2`, etc. may be used, rather than the full field specification (like `$$.i`). A non-terminal or a token may be associated with a union field using the `<fieldname>` specification. E.g.,

```
%token <i> INT          // token association (deprecated, see below)
       <d> DOUBLE
%type  <i> intExpr      // non-terminal association
```

In the example developed here, note that both the tokens and the non-terminals can be associated with a field of the union. However, as noted before, the lexical scanner does not have to know about all this. In our opinion, it is cleaner to let the scanner do just one thing: scan texts. The *parser*, knowing what the input is all about, may then convert strings like `"123"` to an integer value. Consequently, the association of a union field and a token is discouraged. In the upcoming description of the rules of the grammar this will be illustrated further.

In the `%union` discussion the `%token` and `%type` specifications should be noted. They are used to specify the tokens (terminal symbols) that can be returned by the lexical scanner, and to specify the return types of non-terminals. Apart from `%token` the token indicators `%left`, `%right` and `%nonassoc` may be used to specify the associativity of operators. The tokens mentioned at these indicators are interpreted as tokens indicating operators, associating in the indicated direction. The precedence of

operators is given by their order: the first specification has the lowest priority. To overrule a certain precedence in a certain context, `%prec` can be used. As all this is standard `bisonc++` practice, it isn't further elaborated here. The documentation provided with `bisonc++`'s distribution should be consulted for further reference.

Here is the specification of the calculator's declaration section:

```
%filenames parser
%scanner ../scanner/scanner.h
%lines

%union {
    int i;
    double d;
};

%token  INT
        DOUBLE

%type   <i> intExpr
%type   <d> doubleExpr

%left   '+'
%left   '*'
%right  UnaryMinus
```

In the declaration section `%type` specifiers are used, associating the `intExpr` rule's value (see the next section) to the `i`-field of the semantic-value union, and associating `doubleExpr`'s value to the `d`-field. At first sight this may look complex, since the expression rules must be included for each individual return type. On the other hand, if the union itself would have been used, we would still have had to specify somewhere in the returned semantic values what field to use: less rules, but more complex and error-prone code.

The grammar rules The rules and actions of the grammar are specified as usual. The grammar for our little calculator is given below. There are quite a few rules, but they illustrate various features offered by `bisonc++`. In particular, note that no action block requires more than a single line of code. This keeps the organization of the grammar relatively simple, and therefore enhances its readability and understandability. Even the rule defining the parser's proper termination (the empty line in the `line` rule) uses a single member function call `done()`. The implementation of that function is simple, but interesting in that it calls **Parser::ACCEPT()**, showing that the **ACCEPT()** member can be called indirectly from a production rule's action block. Here are the grammar's production rules:

```
    lines:
        lines
        line
    |
        line
    ;

    line:
        intExpr
        '\n'
        {
            display($1);
        }
    |
        doubleExpr
        '\n'
```

```
        {
            display($1);
        }
    |
        '\n'
        {
            done();
        }
    |
        error
        '\n'
        {
            reset();
        }
    ;

    intExpr:
        intExpr '*' intExpr
        {
            $$ = exec('*', $1, $3);
        }
    |
        intExpr '+' intExpr
        {
            $$ = exec('+', $1, $3);
        }
    |
        '(' intExpr ')'
        {

            $$ = $2;
        }
    |
        '-' intExpr         %prec UnaryMinus
        {
            $$ = neg($2);
        }
    |
        INT
        {
            $$ = convert<int>();
        }
    ;

    doubleExpr:
        doubleExpr '*' doubleExpr
        {
            $$ = exec('*', $1, $3);
        }
    |
        doubleExpr '*' intExpr
        {
            $$ = exec('*', $1, d($3));
        }
    |
        intExpr '*' doubleExpr
        {
            $$ = exec('*', d($1), $3);
        }
```

```
    |
        doubleExpr '+' doubleExpr
        {
            $$ = exec('+', $1, $3);
        }
    |
        doubleExpr '+' intExpr
        {
            $$ = exec('+', $1, d($3));
        }
    |
        intExpr '+' doubleExpr
        {
            $$ = exec('+', d($1), $3);
        }
    |
        '(' doubleExpr ')'
        {
            $$ = $2;
        }
    |
        '-' doubleExpr         %prec UnaryMinus
        {

            $$ = neg($2);
        }
    |
        DOUBLE
        {
            $$ = convert<double>();
        }
    ;
```

The above grammar is used to implement a simple calculator in which integer and real values can be negated, added, and multiplied, and in which standard priority rules can be circumvented using parentheses. The grammar shows the use of typed nonterminal symbols: `doubleExpr` is linked to real (double) values, `intExpr` is linked to integer values. Precedence and type association is defined in the parser's definition section.

The Parser's header file Various functions called from the grammar are defined as template functions. `Bisonc++` generates various files, among which the file defining the parser's class. Functions called from the production rule's action blocks are usually member functions of the parser, and these member functions must be declared and defined. Once `bisonc++` has generated the header file defining the parser's class it will not automatically rewrite that file, allowing the programmer to add new members to the parser class. Here is the `parser.h` file as used for our little calculator:

```
#ifndef Parser_h_included
#define Parser_h_included

#include <iostream>
#include <sstream>
#include <bobcat/a2x>

#include "parserbase.h"
#include "../scanner/scanner.h"


#undef Parser
```

```
class Parser: public ParserBase
{
    std::ostringstream d_rpn;
    // $insert scannerobject
    Scanner d_scanner;

    public:
        int parse();

    private:
        template <typename Type>
            Type exec(char c, Type left, Type right);
        template <typename Type>
            Type neg(Type op);
        template <typename Type>
            Type convert();

        void display(int x);
        void display(double x);
        void done() const;
        void reset();
        void error(char const *msg);
        int lex();
        void print();

        static double d(int i);

    // support functions for parse():

        void executeAction(int d_ruleNr);
        void errorRecovery();
        int lookup(bool recovery);
        void nextToken();
};


inline double Parser::d(int i)
{
    return i;
}

template <typename Type>
Type Parser::exec(char c, Type left, Type right)
{
    d_rpn << " " << c << " ";
    return c == '*' ? left * right : left + right;
}

template <typename Type>
Type Parser::neg(Type op)
{
    d_rpn << " n ";
    return -op;
}

template <typename Type>
Type Parser::convert()
{
    Type ret = FBB::A2x(d_scanner.YYText());
```

```
    d_rpn << " " << ret << " ";
    return ret;
}

inline void Parser::error(char const *msg)
{
    std::cerr << msg << std::endl;
}

inline int Parser::lex()
{
    return d_scanner.yylex();
}

inline void Parser::print()
{}

#endif
```

23.9.2.2 The 'flex' specification file

The flex-specification file used by our calculator is simple: blanks are skipped, single characters are returned, and numeric values are returned as either `Parser::INT` or `Parser::DOUBLE` tokens. Here is the complete `flex` specification file:

```
%{
    #define _SKIP_YYFLEXLEXER_
    #include "scanner.ih"

    #include "../parser/parserbase.h"
%}

%option yyclass="Scanner" outfile="yylex.cc" c++ 8bit warn noyywrap
%option debug

%%

[ \t]                       ;
[0-9]+                      return Parser::INT;

"."[0-9]*                   |
[0-9]+("."[0-9]*)?          return Parser::DOUBLE;

.|\n                        return *yytext;

%%
```

23.9.2.3 Generating code

The code is generated in the same way as with `bison` and `flex`. In order to have `bisonc++` generate the files `parser.cc` and `parser.h`, issue the command:

```
    bisonc++ -V grammar
```

The option `-V` will generate the file `parser.output` showing information about the internal structure of the provided grammar, among which its states. It is useful for debugging purposes, and can be left

out of the command if no debugging is required. `Bisonc++` may detect conflicts (shift-reduce conflicts and/or reduce-reduce conflicts) in the provided grammar. These conflicts may be resolved explicitly using disambiguation rules or they are 'resolved' by default. A shift-reduce conflict is resolved by shifting, i.e., the next token is consumed. A reduce-reduce conflict is resolved by using the first of two competing production rules. `Bisonc++` uses identical conflict resolution procedures as `bison` and `bison++`.

Once a parser class and parsing member function has been constructed `flex` may be used to create a lexical scanner (in, e.g., the file `yylex.cc`) using the command

```
flex -I lexer
```

On Unix systems a command comparable to:

```
g++ -o calc -Wall *.cc -s
```

can be used to compile and link both the source of the main program and the generated sources. Finally, here is a source file in which the `main()` function and the parser object is defined. The parser features the lexical scanner as one of its data members:

```
#include "parser/parser.h"

using namespace std;

int main()
{
    Parser parser;

    cout << "Enter (nested) expressions containing ints, doubles, *, + and "
            "unary -\n"
            "operators. Enter an empty line to stop.\n";

    return parser.parse();
}
```

`Bisonc++` can be downloaded from `http://bisoncpp.sourceforge.net/`. It requires the `bobcat` library, which can be downloaded from `http://bobcat.sourceforge.net/`.

23.9.3 Using polymorphic semantic values with Bisonc++

Below the way Bisonc++ may use a polymorphic semantic value is discussed. The approach discussed below is a direct result of a suggestion initially brought forward by Dallas A. Clement in September 2007.

One may wonder why a `union` is still used by Bisonc++ as **C++** offers inherently superior approaches to combine multiple types into one type. The **C++** way to combine types into one type is by defining a polymorphic base class and a series of derived classes implementing the alternative data types. Bisonc++ supports The `union` approach for backward compatibility reasons: both **bison**(1) and **bison++** support the `%union` directive.

The alternative to using a `union` is using a polymorphic base class. This class will be developed below as the class `Base`. Since it's a polymorphic base class it should have the following characteristics:

- Its destructor must be virtual;
- Objects of the derived classes may be obtained using a pure virtual `clone()` member implementing a so-called *virtual constructor* (cf. the *virtual constructor* design pattern, *Gamma et al.* (1995));

- Several convenient utility members may be provided: a pure virtual `insert()` member and an overloaded `operator«()` were defined to allow derived objects to be inserted into `ostream` objects.

The class **Base** has the following interface:

```
#ifndef INCLUDED_BASE_
#define INCLUDED_BASE_

#include <iosfwd>

class Base
{
    public:
        virtual ~Base();
        virtual Base *clone() const = 0;
        virtual std::ostream &insert(std::ostream &os) const = 0;
};

inline Base::~Base()
{}

inline std::ostream &operator<<(std::ostream &out, Base const &obj)
{
    return obj.insert(out);
}

#endif
```

The alternatives as defined by a classical `union` are now defined by classes derived from the class `Base`. For example:

- Objects of the class `Int` contain `int` values. Here is its interface (and implementation):

```
#ifndef INCLUDED_INT_
#define INCLUDED_INT_

#include <ostream>

#include <bobcat/a2x>

#include "../base/base.h"

class Int: public Base
{
    int d_value;

    public:
        Int(char const *text);
        Int(int v);
        virtual Base *clone() const;
        int value() const;               // directly access the value
        virtual std::ostream &insert(std::ostream &os) const;
};

inline Int::Int(char const *txt)
:
    d_value(FBB::A2x(txt))
```

```
{}

inline Int::Int(int v)
:
    d_value(v)
{}

inline Base *Int::clone() const
{
    return new Int(*this);
}

inline int Int::value() const
{
    return d_value;
}

inline std::ostream &Int::insert(std::ostream &out) const
{
    return out << d_value;
}

#endif
```

- Objects of the class `Text` contain text. These objects can be used, e.g., to store the names of identifiers recognized by a lexical scanner. Here is the interface of the class `Text`:

```
#ifndef INCLUDED_TEXT_
#define INCLUDED_TEXT_

#include <string>
#include <ostream>

#include "../base/base.h"

class Text: public Base
{
    std::string d_text;

    public:
        Text(char const *id);
        virtual Base *clone() const;
        std::string const &id() const;          // directly access the name.
        virtual std::ostream &insert(std::ostream &os) const;
};

inline Text::Text(char const *id)
:
    d_text(id)
{}

inline Base *Text::clone() const
{
    return new Text(*this);
}

inline std::string const &Text::id() const
{
    return d_text;
```

```
    }

    inline std::ostream &Text::insert(std::ostream &out) const
    {
        return out << d_text;
    }

    #endif
```

The polymorphic `Base`, however, can't be used as the parser's semantic value type:

- A `Base` class object cannot contain derived class's data members, so plain `Base` class objects cannot be used to store the parser's semantic values.
- It's not possible to define a `Base` class reference as a semantic value either as containers cannot store references.
- Finally, the semantic value should not be a pointer to a `Base` class object. Although a pointer would offer programmers the benefits of the polymorphic nature of the `Base` class, it would also require them to keep track of all memory used by `Base` objects, thus countering many of the benefits of using a polymorphic base class.

To solve the above problems, a *wrapper class* `Semantic` around a `Base` pointer is used. The wrapper class will take care of the proper destruction of objects accessed via its `Base` pointer data member and it will offer facilities to access the proper derived class. The interface of the class `Semantic` is:

```
#ifndef INCLUDED_SEMANTIC_
#define INCLUDED_SEMANTIC_

#include <ostream>

#include "../base/base.h"

class Semantic
{
    Base *d_bp;

    public:
        Semantic(Base *bp = 0);             // Semantic will own the bp
        Semantic(Semantic const &other);
        ~Semantic();

        Semantic &operator=(Semantic const &other);

        Base const &base() const;

        template <typename Class>
        Class const &downcast();

    private:
        void copy(Semantic const &other);
};

inline Base const &Semantic::base() const
{
    return *d_bp;
}

inline Semantic::Semantic(Base *bp)
```

```
:
    d_bp(bp)
{}

inline Semantic::~Semantic()
{
    delete d_bp;
}

inline Semantic::Semantic(Semantic const &other)
{
    copy(other);
}

inline Semantic &Semantic::operator=(Semantic const &other)
{
    if (this != &other)
    {
        delete d_bp;
        copy(other);
    }
    return *this;
}

inline void Semantic::copy(Semantic const &other)
{
    d_bp = other.d_bp ? other.d_bp->clone() : 0;
}

template <typename Class>
inline  Class const &Semantic::downcast()
{
    return dynamic_cast<Class &>(*d_bp);
}

inline std::ostream &operator<<(std::ostream &out, Semantic const &obj)
{
    if (&obj.base())
        return out << obj.base();

    return out << "<UNDEFINED>";
}

#endif
```

`Semantic` is a slightly more 'complete' class than `Base` and its derivatives, since it contains a pointer which must be handled appropriately. So it needs a copy constructor, an overloaded assignment operator and a destructor. Apart from that, it supports members to obtain a reference to the base class. This reference is then used by the overloaded `operator«()` to allow insertion into streams of objects of classes derived from `Base`. It also offers a small member template returning a reference to a derived class object from the semantic value's `Base` class pointer. This member effectively implements (and improves) the type safety that is otherwise strived for by typed nonterminals and typed tokens (using the `%type` directive).

23.9.3.1 The parser using a polymorphic semantic value type

In Bisonc++'s grammar specification `%stype` will of course be `Semantic`. A simple grammar is defined for this illustrative example. The grammar expects input according to the following rule:

```
rule:
    IDENTIFIER '(' IDENTIFIER ')' ';'
|
    IDENTIFIER '=' INT ';'
;
```

The rule's actions simply echo the received identifiers and `int` values to `cout`. Here is an example of a decorated production rule:

```
IDENTIFIER '=' INT ';'
{
      cout << $1 << " " << $3 << endl;
}
```

Alternative actions could easily be defined, e.g., using the `Base::downcast()` member:

```
IDENTIFIER '=' INT ';'
{
    int value = $3.downcast<Int>().value();
}
```

Bisonc++'s parser stores *all* semantic values on its semantic values stack (irrespective of the number of tokens that were defined in a particular production rule). At any time *all* semantic values associated with previously recognized tokens are available in an action block. Once the semantic value stack is reduced, the `Semantic` class takes care of the proper destruction of the objects pointed to by the `Semantic` data member `d_bp`.

The scanner must of course be able to access the parser's data member representing the most recent semantic value. This data member is available as the parser's data member `d_val__`, which can be offered to the scanner object at its construction time. E.g., with a scanner expecting an `STYPE__ &` the parser's constructor could simply be defined as:

```
inline Parser::Parser()
:
    d_scanner(d_val__)
{}
```

23.9.3.2 The scanner using a polymorphic semantic value type

As discussed in the previous section the scanner must have access to the parser's `d_val__` data member. Therefore the Scanner class may define a `Semantic &d_semval` member, which is initialized to `Semantic d_val__` which is passed to the Scanner's constructor via the constructor's `semval` parameter:

```
inline Scanner::Scanner(Parser::STYPE__ &semval)
:                       // or: Semantic &semval
    d_semval(semval)
{}
```

The scanner (generated by **flex**(1)) recognizes input patterns, returns Parser tokens (e.g., Parser::INT), and returns a semantic value when applicable. E.g., when recognizing a `Parser::INT` the rule is:

```
{
    *d_semval = new Int(yytext);
    return Parser::INT;
}
```

Note that, as the `Semantic` constructor expects but one argument, automatic promotion from `Base *` to `Semantic` can be used in the assignments to `*d_semval`.

The `IDENTIFIER`'s semantic value is obtained as follows:

```
[a-zA-Z_][a-zA-Z0-9_]* { *d_semval = new Text(yytext); return Parser::IDENTIFIER; }
```

Index