ℵ Programming Language

C++ Programming Interface

CONTENTS

Preface

This manual is part of the *Aleph Programming Language Series*, a multi volume set that describes the programming environment of the **Aleph** system. The entire set contains 4 volumes :

Volume 0 - Aleph Installation Guide is the distribution installation manual.

Volume 1 - Aleph Programmer Guide is the first volume of this set. It is both an introduction and an advanced guide for the the developer.

Volume 2 - Aleph Library Reference is the second volume of this set. It is a complete description of the Aleph standard library.

Volume 3 - Aleph Cross Debugger is the third volume of this set. It is a reference manual to develop and debug Aleph programs.

Volume 4 - Aleph C++ API is the fourth volume of this set. It is a reference manual of the C++ Application Programming Interface (API).

The Aleph programming language

Aleph is a multi-threaded functional programming language with dynamic symbol bindings that support the object oriented paradigm. **Aleph** features a state of the art runtime engine that supports both 32 and 64 bits platforms. **Aleph** comes with a rich set of libraries that are designed to be platform independent. **Aleph** is a free software. A flexible license has been designed for both individuals and corporations. Everybody is encouraged to use, distribute and/or modify the aleph engine for any purpose.

Features

The **Aleph** engine is written in C++ and provides runtime compatibility with it. Such compatibility includes the ability to instantiate C++ classes, use virtual methods and raise or catch exceptions. A comprehensive API has been designed to ease the integration of foreign libraries.

- **Builtin objects**
  More than 50 reserved keywords and predicates. Various containers like list, vector, hash table, bitset, and graphs.

- **Functional programming**
  Support for *lambda expression* with explicit closure. Symbol scope limitation with *gamma expression*. Form like notation with an easy block declaration.

- **Object oriented**
  Single inheritance object mechanism with dynamic symbol resolution. Native class deriva-tion and method override. Static class data member and methods.

- **Multi-threaded engine**
  True multi-threaded engine with automatic object protection mechanism against concurrent access. Read and write locking system and thread activation via condition objects.

- **Original regular expression**
  Builtin regular expression engine with group matching, exact or partial match and substitu-tion.

Aleph is a core language and libraries. The libraries are a specific set of classes and functions which are structured per application domains. **Aleph** is delivered with a set of standard libraries.

- **aleph-sys**
  The `aleph-sys` library is the system calls library. Standard classes and functions are provided to interact with the running machine.

- **aleph-sio**
  The `aleph-sio` library is the standard input/output All input/output operations are per-formed with this library.

- **aleph-net**
  The `aleph-net` library is the networking library. The library is based on the standard *Internet Protocol* and provides various classes to manipulates IP address, client or server sockets.

- **aleph-www**
  The `aleph-www` library is the World Wide Web library. The library provides various classes that ease the development of web applications or CGI scripts.

- **aleph-txt**
  The `aleph-txt` library is the text processing library. The library provides various func-tions and classes that ease text manipulation. Sorting data, computing message digest and formatting table is among others, features available in this library.

- **aleph-odb**
  The `aleph-odb` library is the object database library. The library provides several objects that can be used to design a database. A client is also provided to directly access the database contents.

Aleph provides *extensions*. An extension is a library or an application which is not installed by default. The user selects during the installation process which extension is needed. For example, the static version of the interpreter is an extension.

Aleph engine

Aleph is an interpreted language. When used interactively, commands are entered on the command line and executed when a complete and valid syntactic object has been constructed. Alternatively, the interpreter can execute a source file. **Aleph** does not have a garbage collector. **Aleph** operates with a lazy, scope based, object destruction mechanism. Each time an object is no longer visible, it is destroyed automatically. At this time, the **Aleph** interpreter is unable to reclaim memory with circular structures. This is a well known problem when using a reference count mechanism. In the future, the **Aleph** engine will provide some mechanisms to resolve this problem.

Flexible Distribution

Aleph is a free software. A flexible license model encourages individuals or corporations to use, copy, modify and/or distribute this software. **Aleph** is designed by software professionals. Quality is one the driving force of the development effort. This is reflected in this distribution by the extensive documentation. A large test suite is used to assess the quality of the distribution. Right now, the engine has been successfully tested on most Linux platforms, Free BSD and Solaris.

License

Aleph is a free software. It can be used, modified and distributed by anybody for personal or commercial use. The only restriction is altering the copyright notice associated with the material. Individual or corporation are permitted to use, include or modify the **Aleph** engine. All material developed with the **Aleph** language belongs to their respective copyright holder.

CHAPTER 1
Introduction

This chapter is an introduction to the general organization of the **Aleph** C++ *application programming interface* or API. We start by looking at the various levels of the API and then describes the general presentation of the C++ header files. The reader needs to absorb some materials before beeing able to write its own class. For the impatient, we recommend to read this chapter and the next one. A simple example is given at the end of chapter 2.

1.1   The header files

All **Aleph** API files are structured into several directories, each of them dedicated to a specific functions. All directories are generally located below an **aleph** specific directory or for some distributions under the standard /usr/include or /usr/local/include directories for a UNIX platform.

- **plt**
  The platform independent header files.

- **std**
  The standard object header files.

- **eng**
  The **Aleph** engine header files.

Additional directories are included. One for each library and extension.

1.1.1   The platform headers

The plt directory contains the platform independant headers files. The files are low level functions and should be the last thing to use. Their use will be covered later.

1.1.2   The standard objets headers

The std directory contains the standard object headers files. This directory is the primary source of files. It contains the most important header, named **Object**. The **Object** header is covered in the next chapter.

1.1.3   The interpreter engine headers

The eng directory contains the interpreter engine header files. This directory is as important as the standard object one. It contains the **Interp** object, which is the interpreter by itself.

1.2   Conventions

The general structure of the **Aleph** API rely on various conventions and rules. These rules have been carefully designed to ensure portability accross platforms as well as a smooth support for 32 and 64 bits platforms.

1.2.1   Header file extension

All header files use the hpp extension. These header files only reference **aleph** functions or classes. If there is one rule to keep in mind, it is this one. **There is no platform header files**. This rule provides the flexibility to design a system which is platform independant. Because of the Makefile structure, any attempt to use a platform header will cause a compilation error. If one take a look at the source tree, (s)he will find another header type with extension hxx. This type of header are the one which contains the platform dependant headers. However, this type of headers is never exported to the API.

1.2.2   Aleph namespace

All functions and classes belong to the namespace aleph. This namespace should be used only for software code which is supposed to be part of the **Aleph** distribution. If you plan to develop your own piece of code, please use another namespace.

1.2.3   Makefile

There is no particular rules for Makefile, unless the software beeing develop is supposed to be integrated into the distribution.

1.3   Types, declaration and memory

The basic types used everywhere are defined in the plt/ccnf.hpp header file. This file contains also some information about your operating system, the processor type and the software revision. It is strongly suggested to take a look at it. The basic types are defined inside the aleph namespace.

```
typedef unsigned char       t_byte;
typedef unsigned short      t_word;
typedef unsigned int        t_quad;
typedef unsigned long long  t_octa;
typedef long long           t_long;
typedef double              t_real;
```

The special type t_size is determined automatically according to the platform. A 32 bits platform has:

```
typedef unsigned int        t_size;
```

while a 64 bits platform has:

```
typedef unsigned long       t_size;
```

It is by far the `t_size` type which is the most important. This type is automatically determined by your platform configuration. You should almost never use the `int` type, except for resource description. The `long` type is most of the time appropriate and `t_long` required when a 64 bits width needs to be forced. Since most compiler adopt the *LP64* model, the `long` type seems to be a good choice. Note that this header file is automatically included in the **Object** file.

1.3.1 Character and strings

The `char` type is the prefered type for a character. The implementation uses the `const char*` for the c-strings. The standard object library provides a **String** class that take care of string manipulation. On the other hand, the basic c-string declaration is inevitable when dealing with literal string. The `t_byte` type is appropriate for *unsigned character*.

1.3.2 Memory allocation

We will talk a lot about memory allocation later. At this stage, note that `new` and `delete` are the best way to operate. You should almost never use the low level memory allocation since it might conflict with the *memory tracer* and worse with the shared libaries memory cleanup subsystem. Obviously, call to `malloc` and `free` are prohibited. In fact, these functions are not exported by the API. If an object is generated by a function or method and need to be destroyed, a call to `delete` is the way to go, nothing else will work. **As a matter of facts, the object destruction is a more complex subject, which is detailed in chapter 3**.

CHAPTER 2

Eval-Apply and functions

The *eval-apply* mechanism is the central runtime operation of the **Aleph** engine. Depending on the nature of the form to evaluate, one strategy or another is used to perform the *eval-apply* process. It is interesting to note that a complete program execution is merely an *eval-apply* loop.

2.1 Interpreter eval-apply

The **Aleph** interpreter rely on a mechanism called an *eval-apply* loop. For each form being evaluated, the first object evaluation method is called. The returned object is eventually used to apply one or several evaluated arguments. We describe in detail how this mechanism works with various example.

2.1.1 Various eval-apply

There are several kinds of *eval-apply* strategy that are used depending on the nature of the form.

- *add + 1 2*
  This is the simplest form to evaluate. The form consist of a list of objects. The first object is evaluated and the rest is transmitted to the evaluated object during the *apply* phase. Our first example will cover this type of form.

- *Boolean true*
  This form is an object construction form. Its evaluation is slightly different to the previous one. This type of form is covered in the next chapter.

- *aleph:sys:sleep 1000*
  This form is similar to the first one, except that it uses a qualified name instead of the a lexical name.

- *str:split ":"*
  This form is a method call with argument. It's evaluation is described in the next chapter.

- *println "hello world"*
  This form is a special form. Its evaluation is described in a specific chapter. Forms that uses a reserved keyword are generally special forms. Note that the reserved keywords **const** and **trans** are special, for a special form.

- *+ 1 2*
  This form is a generic operating form. Its evaluation is described in a specific chapter.

2.1.2   The simplest eval-apply

The simplest example to consider is the introductory form add 1 2. This form consists of three objects. The first one is a lexical object, the second and third are literals. When the form is evaluated several steps are performed. The form is represented by three *cons cells*. The *car* contains the lexical object. The *cdr* points to another *cons cell*, those *car* is an Integer object. The *cdr* of the second *cons cell* points to the third argument, those *car* is also an Integer object. The last *cdr* is *nil* (null pointer) to indicate the end of the form. This form is automatically built by the **Aleph** reader. During the evaluation process, several operations take place, as described below.

- Evaluates the first object in the form. The object is a lexical object represented by add. Such evaluation is done with the interpreter by calling the object virtual method **eval**. In that particular case, the lexical evaluates to a function object.

- Apply the first evaluated object with the rest of the form. For that particular example, the first object has been evaluated as a function object. The virtual **apply** method is called with the *cdr* of the form. The apply method returns the result of such application. In that particular case, the add function method returns an Integer object, those value is 3.

- What happen inside the **apply** method is implementation dependent. However, for that particular example, the add method evaluates each argument object, check that they are valid integers, compute the sum, and returns a newly created integer object with the result.

This simple example illustrates the fundamental mechanics of the **Aleph** engine. Note that the process we have described here is simply the evaluation process of a *cons cell*. We will come back later on this.

2.1.3   Runnable object

The **aleph** interpreter is a *runnable* object. A *runnable* object is a special object that carry several methods. Our first example do not need it, but the object is part of the API. A typical use of the *runnable* object is within the println builtin function (special form). Such function get the standard output stream from the *runnable* object. The *runnable* object is defined in the standard library as **Runnable**.

2.1.4   Evaluation nameset

An *eval-apply* process is done within an *evaluation nameset*. The *evaluation nameset* is the point at which a symbol or an argument is evaluated. The first example does use the evaluation nameset to resolve the symbol *add*. The *nameset* object is defined in the standard library as **Nameset**.

2.1.5   Object eval-apply methods

The **Object** class contains several overloaded **eval** and **apply** methods. We are here only concerned with the simple one.

2.2   The add function

The C++ implementation of the **add** function declaration is given below. The first argument is the *runnable* object. The second argument is the *evaluation nameset* object. The third argument is the argument list. At this stage of the call, the symbol add has been evaluated (we will how later), and the function is called with the last two arguments as a result of an apply call.

```
Object* example_add (Runnable* robj, Nameset* nset, Cons* args) {
  // evaluate the arguments in a vector
  Vector* argv = Vector::eval (nset, args);
  // compute the result
  long result = argv->getint (0) + argv->getint (1);
  // generate result
  return new Integer (result);
}
```

As it can be noticed, this implementation is quite simple and will work. Unfortunately, it will also leak, since the argument vector is not destroyed. Note also that an exception might also happen. A perfect implementation should enclose the computation in a *try-catch* block and destroy the argument vector. The complete implementation, ready to compile, is given below, with the appropriate include file and namespace. This example is available in the `exp/api` directory as `Add.cpp`.

```
#include "Vector.hpp"
#include "Integer.hpp"

namespace example {
  // use the aleph namespace
  using namespace aleph;

  // add function implementation
  Object* example_add (Runnable* robj, Nameset* nset, Cons* args) {
    // evaluate the arguments in a vector
    Vector* argv = Vector::eval (robj, nset, args);
    try {
      // compute the result
      long result = argv->getint (0) + argv->getint (1);
      // clean the vector
      delete argv;
      // generate result
      return new Integer (result);
    } catch (...) {
       delete argv;
       throw;
    }
  }
}
```

2.3   Putting all together

Once the `add` function has been implemented, it must be registered within the interpreter. There are various ways to do so. The description below uses the standard **Aleph** mechanism with shared libraries. Once the library will be built, it is going to be possible to test the function.

2.3.1   Shared library entry point

The first thing to do, in order to create a shared library, is to define a unique entry point that is called when the library is loaded. This entry point takes care of registering the library symbols. The **Aleph** convention uses names in the form of *dli_namespace_library*. The plus (+) and and minus (-) characters are automatically re-mapped to underscore character. For example, the library

`aleph-sys` has the entry point `dli_aleph_sys`. Our example uses the namespace example, so a valid entry point would be `dli_example` or preferably `dli_example_api`. The entry point must be mapped as a "C" name to avoid *name mangling*. It is also part of the **Aleph** methodology to break the call in two pieces. One is the "C" entry point and the other is the "C++" entry point.

```
#include "Libapi.hpp"
#include "Apicalls.hpp"
#include "Function.hpp"

namespace example {
  // use the aleph namespace
  using namespace aleph;

  // initialize the api library
  Object* init_example_api (Interp* interp, Vector* argv) {
    // make sure we are not called from something crazy
    if (interp == nilp) return nilp;

    // create the api nameset
    Nameset* api = interp->mknset ("api", interp->getgset ());

    // bind the add function
    api->symcst ("add", new Function (api_add));

    // not used but needed
    return nilp;
  }
}

extern "C" {
  aleph::Object* dli_example_api (aleph::Interp* interp,
   aleph::Vector* argv) {
    return example::init_example_api (interp, argv);
  }
}
```

2.3.2 Compiling everything

The compilation is quite simple. The library generation is a little more tricky. The previous example is demonstrated with GCC and assume that the **Aleph** include file are located in the `/usr/local/include/aleph` directory.

```
zsh > g++ -I. -I/usr/local/include/aleph -g
        -fPIC -D_REENTRANT -c *.cpp
zsh > g++ -shared -o libexample-api.so *.o
        -L/usr/local/lib -laleph-eng
```

These commands should work fine. However, depending on your system, some adjustments might be needed. You should look at the **Aleph** compilation process to get a better idea of what is going on. Some system might require all libraries. More compiler flags can be used, and should be used.

2.3.3 Testing the result

With the library ready to use, it is possible to run an **Aleph** session and see what is happening.

```
zsh> aleph
aleph> interp:library "example-api"
aleph> println (add 1 2)
3
aleph> C-d
zsh >
```

This is it. If it does not work, you should check the LD_LIBRARY_PATH environment variable. In last resort, the directory exp/api contains a Makefile designed to work correctly, but it won't install the library.

CHAPTER 3
Object Class

The **Object** class is the *pillar* of the **Aleph** engine. The class defines the base methods that are used during the *eval-apply* process. The class provides also the methods to control the object locking as well as the object reference count.

3.1 Object evaluation

The simplest object evaluation is the one that takes an object and returns an object. Most of the time this evaluation is *reflexive*. That is the calling object is returned. In this case, the object is said to be *self evaluated*.

3.1.1 Object eval-apply methods

The **Object** class contains several overloaded **eval** and **apply** methods. The simplest one requires only a *runnable* and *nameset* objects.

```
class Object {
  ... many declarations

  /// evaluate an object in the current nameset
  /// @param robj   the current runnable
  /// @param nset   the current nameset
  virtual Object* eval (class Runnable* robj, class Nameset* nset);

  /// apply an object with a set of arguments
  /// @param robj   the current runnable
  /// @param nset   the current nameset
  /// @param args   the arguments to apply
  virtual Object* apply (class Runnable* robj, class Nameset* nset,
                         class Cons* args);
}
```

3.1.2 Eval default implementation

The default implementation for the **eval** method is to return the calling object (aka `this`). As mentioned earlier, such behavior is called *self evaluation*. Most of the objects evaluates to themselves. This include all literal objects such like `Integer` or *String*.

```
  // evaluate an object in the current nameset
  Object* Object::eval (Runnable* robj, Nameset* nset) {
    return this;
  }
}
```

It is remarkable to notice how such simple implementation can be so powerful.

3.1.3 Qualified name evaluation

The evaluation of a qualified name is a repetitive process that goes for each lexical element of that name. For example, the qualified name `hello:world` is evaluated first by evaluating `hello` which in turn evaluates `world`. Since, the evaluation by name is too costly in terms of stringg comparision, the qualified name evaluation is done by a `quark` mechanism.

3.1.4 Quark definition

A quark is unique integer representation for a given string. The term `quark` comes from the *X Window System* which has a similar mechanism. For a given string, a quark is constructed with the static string method `intern`.

```
  static const long QUARK_TOSTRING = String::intern ("to-string");
```

When a qualified name is constructed by the lexical analyzer, an equivalent quark representation is computed automatically. This process makes now the qualified name evaluation a straight-forward recursive system that involves just integer comparision.

3.2 Static construction

In order for the interpreter to construct a new object, the class has to provide a static method that returns such object based on an argument vector.

```
  /// generate a new boolean
  /// @param argv the argument vector
  static Object* mknew (Vector* argv);
```

The vector argument contains *evaluated object*. It is up to the implementation to call the appropriate constructor, depending on the arguments type and values.

3.2.1 Argument vector interface

The **Vector** class provides several method that ease the transformation between object and native types.

- `length`
  returns the numbre of elements in this vector

- `getint`
  returns a native integer by index

- `getbool`
  returns a native boolean by index

- `getreal`
  returns a native real (double) by index

- `getchar`
  returns a native character by index

- `getstring`
  returns a string object by index

As an example, we reproduce here the implementation of the **BitSet** class.

```
Object* BitSet::mknew (Vector* argv) {
  long argc = (argv == nilp) ? 0 : argv->length ();
  if (argc == 0) return new BitSet;
  if (argc == 1) {
    long size = argv->getint (0);
    return new BitSet (size);
  }
  throw Exception ("argument-error", "too many argument for bitset");
}
```

This implementation provides two ways to create a **bitSet** object. Without argument, the argument vector length is null and the method return a new object by calling the default constructor. With one argument, a new bit set is created with a specific size. The other cases throw an exception. Note that an exception can be raised by the `getint` method if the object argument cannot be mapped to an integer value.

Another way to access the argument object is by performing directly *dynamic casting*. We left the code fragment below of the **Character** class as an excercise.

```
Object* Character::mknew (Vector* argv) {
  if ((argv == nilp) || (argv->length () == 0)) return new Character;
  if (argv->length () != 1)
    throw Exception ("argument-error",
                     "too many argument with character constructor");
  // try to map the character argument
  Object* obj = argv->get (0);
  if (obj == nilp) return new Character;

  // try an integer object
  Integer* ival = dynamic_cast <Integer*> (obj);
  if (ival != nilp) return new Character (ival->tointeger ());

  // try a character object
  Character* cval = dynamic_cast <Character*> (obj);
  if (cval != nilp) return new Character (*cval);

  // try a string object
  String* sval = dynamic_cast <String*> (obj);
  if (sval != nilp) return new Character (*sval);

  // illegal object
  throw Exception ("type-error", "illegal object with character"
                   obj->repr ());
}
```

3.3   A simple object

Writing a new C++ object that is usable by the **Aleph** engine is quite simple. The following example implements the behavior of a *boolean*. The class is called `Boolean` and is available in the standard library. We implement here a simple version without qualified name support, but this minimal example will compile and can be constructed in the interpreter.

```
#ifndef  EXAMPLE_BOOLEAN_HPP
#define  EXAMPLE_BOOLEAN_HPP

#ifndef  ALEPH_OBJECT_HPP
#include "Object.hpp"
#endif

namespace example {
  // use the aleph namespace
  using namespace aleph;

  /// The example::Boolean class
  class Boolean : public Object {
  private:
    /// the native boolean
    bool d_value;

  public:
    /// create a new default boolean
    Boolean (void);

    /// create a boolean by value
    Boolean (const bool value);

    /// @return the class name
    String repr (void) const;

    /// create a new boolean
    static Object* mknew (Vector* argv);
  }
}
#endif
```

This is the minimal declaration that is needed to compile the boolean example. Note that the default constructor is not really needed, but it is given here for illustration purpose.

3.3.1   The default constructor

The default constructor is not really needed here, but it is generally wise to provide a default implementation. Remember that a default constructor is one of the six default implementation provided by the compiler for a given class. It is a good practice to have basic things under control.

```
namespace example {
  // create a default boolean
  Boolean::Boolean (void) {
    d_value = false;
  }
```

```
  // create a boolean by value
  Boolean::Boolean (const bool value) {
    d_value = value;
  }
}
```

3.3.2 The representation method

The **repr** method is one of the mandatory method for an object derivation since it is defined as *virtual pure* in the `Object` base class. The method returns a string representation of that class, that is here the string `"Boolean"`.

```
namespace example {
  // return the class name
  String Boolean::repr (void) const {
    return "Boolean";
  }
}
```

3.3.3 The static constructor

The implementation for the static constructor is trivial. We accept only 0 or one argument and create a new boolean object.

```
namespace example {
  // create a boolean in a generic way
  static Object* Boolean::mknew (Vector* argv) {
    long argc = (argv == nilp) ? 0 : argv->length ();
    if (argc == 0) return new Boolean;
    if (argc == 1) {
      long value = argv->getbool (0);
      return new Boolean (value);
    }
    throw Exception ("argument-error", "too many argument for boolean");
  }
}
```

3.3.4 First compilation

If we assume that the standard **Aleph** headers are located under the `/usr/local/include/aleph/std`, we can compile the previous example, assuming we are using `gcc`.

```
zsh > gcc -D_REENTRANT -I. -I/usr/local/include/aleph/std
          -I/usr/local/include/aleph/std -o Boolean.o -c Boolean.cpp
```

The flags for compiling in debug mode, all warnings, no standard include files, etc. can be used as well. We will see in the next chapter how to use the standard **Aleph** makefile to develop a complete library. If you plan to use your own build system, a complete list of directives and recommendation is given in the next chapter as well. If you plan to contribute to the **Aleph** distribution, ther are more constraints attached to the build process which are described in a specific *Contributing to Aleph* chapter.

APPENDIX A
Boolean example

A.1 Boolean example header

```
#ifndef  ALEPH_BOOLEAN_HPP
#define  ALEPH_BOOLEAN_HPP

#ifndef  ALEPH_LITERAL_HPP
#include "Literal.hpp"
#endif

#ifndef  ALEPH_SERIAL_HPP
#include "Serial.hpp"
#endif

namespace aleph {

  class Boolean : public Literal, public Serial {
  private:
    /// the native boolean representation
    bool d_value;

  public:
    /// create a new default boolean - by default it is false
    Boolean (void);

    /// create a new boolean from a native boolean
    /// @param value the value to create
    Boolean (const bool value);

    /// create a new boolean from a string
    /// @param value the value to convert
    Boolean (const String& value);

    /// copy constructor for this boolean
    /// @param that the boolean class to copy
    Boolean (const Boolean& that);
```

```
/// @return the class name
String repr (void) const;

/// @return a literal representation of this boolean
String toliteral (void) const;

/// @return a string representation of this boolean
String tostring (void) const;

/// @return a clone of this boolean
Object* clone (void) const;

/// @return the boolean serial code
t_byte serialid (void) const;

/// serialize this boolean to an output stream
/// @param os the output stream to write
void wrstream (class Output& os) const;

/// deserialize a boolean from an input stream
/// @param is the input steam to read in
void rdstream (class Input& is);

/// @return the boolean value of this boolean
bool toboolean (void) const;

/// assign a boolean with a native value
/// @param value the value to assign
Boolean& operator = (const bool value);

/// assign a boolean with a boolean
/// @param value the value to assign
Boolean& operator = (const Boolean& value);

/// compare this boolean with a native value
/// @param value the value to compare
/// @return true if they are equals
bool operator == (const bool value) const;

/// compare this boolean with a native value
/// @param value the value to compare
/// @return true if they are not equals
bool operator != (const bool value) const;

/// compare two booleans
/// @param value the value to compare
/// @return true if they are equals
bool operator == (const Boolean& value) const;

/// compare two boolean
/// @param value the value to compare
/// @return true if they are not equals
bool operator != (const Boolean& value) const;
```

```
    /// evaluate an object to a boolean value
    /// @param robj the current runnable
    /// @param nset the current nameset
    /// @param object the object to evaluate
    static bool evalto (Runnable* robj, Nameset* nset, Object* object);

    /// generate a new boolean
    /// @param argv the argument vector
    static Object* mknew (Vector* argv);

    /// operate this boolean with another object
    /// @param robj   the current runnable
    /// @param type   the operator type
    /// @param object the operand object
    Object* oper (Runnable* robj, t_oper type, Object* object);

    /// set an object to this boolean
    /// @param robj   the current runnable
    /// @param nset   the current nameset
    /// @param object the object to set
    Object* vdef (Runnable* robj, Nameset* nset, Object* object);

    /// apply this boolean with a set of arguments and a quark
    /// @param robj   the current runnable
    /// @param nset   the current nameset
    /// @param quark  the quark to apply these arguments
    /// @param argv   the arguments to apply
    Object* apply (Runnable* robj, Nameset* nset, const long quark,
                   Vector* argv);
  };
}
#endif
```

A.2 Boolean example implementation

```
#include "Input.hpp"
#include "Vector.hpp"
#include "Boolean.hpp"
#include "Exception.hpp"

namespace aleph {

  // the boolean supported quarks
  static const long QUARK_EQL      = String::intern ("==");
  static const long QUARK_NEQ      = String::intern ("!=");
  static const long QUARK_TOSTRING = String::intern ("to-string");

  // create a new boolean - the initial value is false

  Boolean::Boolean (void) {
    d_value = false;
```

```
  }

  // create a boolean from a native value

  Boolean::Boolean (const bool value) {
    d_value = value;
  }

  // create a boolean from a string

  Boolean::Boolean (const String& value) {
    if (value == "false")
      d_value = false;
    else if (value == "true")
      d_value = true;
    else
      throw Exception ("literal-error","illegal boolean value",value);
  }

  // copy constructor for this boolean

  Boolean::Boolean (const Boolean& that) {
    d_value = that.d_value;
  }

  // return the class name

  String Boolean::repr (void) const {
    return "Boolean";
  }

  // return a literal representation of this boolean

  String Boolean::toliteral (void) const {
    return tostring ();
  }

  // return a string representation of this boolean

  String Boolean::tostring (void) const {
    return d_value ? "true" : "false";
  }

  // return a clone of this boolean

  Object* Boolean::clone (void) const {
    return new Boolean (*this);
  }

  // return the boolean serial code

  t_byte Boolean::serialid (void) const {
    return SERIAL_BOOL_ID;
```

```
}

// serialize this boolean

void Boolean::wrstream (Output& os) const {
  rdlock ();
  char c = d_value ? 0x01 : nilc;
  os.write (c);
  unlock ();
}

// deserialize this boolean

void Boolean::rdstream (Input& is) {
  wrlock ();
  char c = is.read ();
  d_value = (c == nilc) ? false : true;
  unlock ();
}

// return this boolean value

bool Boolean::toboolean (void) const {
  rdlock ();
  bool result = d_value;
  unlock ();
  return result;
}

// assign a boolean with a native value

Boolean& Boolean::operator = (const bool value) {
  d_value = value;
  return *this;
}

// assign a boolean with a boolean

Boolean& Boolean::operator = (const Boolean& value) {
  d_value = value.d_value;
  return *this;
}

// compare a boolean with a native value

bool Boolean::operator == (const bool value) const {
  return (d_value == value);
}

// compare two boolean

bool Boolean::operator == (const Boolean& value) const {
  return (d_value == value.d_value);
```

```
}

// compare a boolean with a native value

bool Boolean::operator != (const bool value) const {
  return (d_value != value);
}

// compare two boolean

bool Boolean::operator != (const Boolean& value) const {
  return (d_value != value.d_value);
}

// evaluate an object to a boolean value

bool Boolean::evalto (Runnable* robj, Nameset* nset, Object* object) {
  Object* obj = (object == nilp) ? nilp : object->eval (robj, nset);
  Boolean* val = dynamic_cast <Boolean*> (obj);
  if (val == nilp) throw Exception ("type-error", "nil object to evaluate");
  return val->toboolean ();
}

// create a new boolean in a generic way

Object* Boolean::mknew (Vector* argv) {
  if ((argv == nilp) || (argv->length () == 0)) return new Boolean;
  if (argv->length () != 1)
    throw Exception ("argument-error",
     "too many argument with boolean constructor");
  // try to map the boolean argument
  Object* obj = argv->get (0);
  if (obj == nilp) return new Boolean;

  // try a boolean object
  Boolean* bval = dynamic_cast <Boolean*> (obj);
  if (bval != nilp) return new Boolean (*bval);

  // try a string object
  String* sval = dynamic_cast <String*> (obj);
  if (sval != nilp) return new Boolean (*sval);

  // illegal object
  throw Exception ("type-error", "illegal object with boolean constructor",
   obj->repr ());
}

// operate this boolean with another object

Object* Boolean::oper (Runnable* robj, t_oper type, Object* object) {
  Boolean* bobj = dynamic_cast <Boolean*> (object);
  switch (type) {
  case Object::EQL:
```

```
      if (bobj != nilp) return new Boolean (d_value == bobj->d_value);
      break;
    case Object::NEQ:
      if (bobj != nilp) return new Boolean (d_value != bobj->d_value);
      break;
    default:
      throw Exception ("operator-error", "unsupported boolean operator");
    }
    throw Exception ("type-error", "invalid operand with boolean",
     Object::repr (object));
  }

  // set an object to this boolean

  Object* Boolean::vdef (Runnable* robj, Nameset* nset, Object* object) {
    Boolean* bobj = dynamic_cast <Boolean*> (object);
    if (bobj != nilp) {
      d_value = bobj->d_value;
      return this;
    }
    throw Exception ("type-error", "invalid object with boolean vdef",
     Object::repr (object));
  }

  // apply this boolean with a set of arguments and a quark

  Object* Boolean::apply (Runnable* robj, Nameset* nset, const long quark,
  Vector* argv) {
    // get the number of arguments
    long argc = (argv == nilp) ? 0 : argv->length ();

    // dispatch 0 argument
    if (argc == 0) {
      if (quark == QUARK_TOSTRING) return new String (toliteral ());
    }

    // dispatch one argument
    if (argc == 1) {
      if (quark == QUARK_EQL) return oper (robj, Object::EQL, argv->get (0));
      if (quark == QUARK_NEQ) return oper (robj, Object::NEQ, argv->get (0));
    }

    // call the object method
    return Object::apply (robj, nset, quark, argv);
  }
}
```

APPENDIX B
Object class

```
#ifndef  ALEPH_OBJECT_HPP
#define  ALEPH_OBJECT_HPP

#ifndef  ALEPH_CCNF_HPP
#include "ccnf.hpp"
#endif

namespace aleph {

  /// The Object class is the foundation of the standard object library .
  /// The object class defines only a reference count field which is used
  /// to control the life of a particular object. When an object is created,
  /// the reference count is set to 0. Such object is said to be transient.
  /// The "iref" static method increment the reference count. The "dref"
  /// method decrement and eventually destroy the object. The "cref" method
  /// eventually destroy an object if its reference count is nulll. The object
  /// class is an abstract class. For each derived object, the repr method
  /// is defined to return the class name. Additionally, the object class
  /// defines a set of methods which are used by the runnable to virtually
  /// modify or evaluate an object. There are two sets of methods. The first
  /// set operates directly on the object. The second set operates by name
  /// on the object. Working by name is equivalent to access a member of a
  /// a particular object. The "cdef" method create or set a constant object
  /// to the calling object. The "vdef" method create or set an object to the
  /// calling object. The "eval" method evaluates an object in the current
  /// runnable nameset. The "apply" method evaluates a set of arguments
  /// and apply them to the calling object. It is somehow equivalent to a
  /// function call. When called by name, it is equivalent to a method call.
  /// @author amaury darsch

  class Object {
  public:
    enum t_oper {ADD, SUB, MUL, DIV, MINUS, EQL, NEQ, GEQ, LEQ, GTH, LTH};

  private:
    /// object reference count
    long d_rcount;
```

```
protected:
  /// the shared object structure
  struct s_shared* p_shared;

public:
  /// create a new object
  Object (void);

  /// destroy this object.
  virtual ~Object (void);

  /// @return the class name
  virtual class String repr (void) const =0;

  /// @return an object class name or nil
  static const class String repr (Object* object);

  /// @return a clone of this object
  virtual Object* clone (void) const;

  /// make this object shared
  virtual void mksho (void);

  /// get a read lock for this object
  virtual void rdlock (void) const;

  /// get a write lock for this object
  virtual void wrlock (void) const;

  /// unlock this object
  virtual void unlock (void) const;

  /// @return true if the object is shared
  bool issho (void)
    return (p_shared != nilp);


  /// clear and lock the finalizer
  static void clrfnl (void);

  /// increment the object reference count
  /// @param object the object to process
  static Object* iref (Object* object);

  /// decrement the reference count and destroy the object if null
  /// @param object the object to process
  static void dref (Object* object);

  /// clean this object if the reference count is null
  /// @param object the object to process
  static void cref (Object* object);

  /// decrement the object reference count but do not detroy if null
```

```
/// @param object the object to process
static void tref (Object* object);

/// return true if the object has a reference count of 0 or 1
/// @param object the object to process
static bool uref (Object* object);

/// operate this object with another one
/// @param robj   the current runnable
/// @param type   the operator type
/// @param object the operand object
virtual Object* oper (class Runnable* robj, t_oper type, Object* object);

/// set an object as a const object
/// @param robj   the current runnable
/// @param nset   the current nameset
/// @param object the object to set
virtual Object* cdef (class Runnable* robj, class Nameset* nset,
Object* object);

/// set an object as a const object by quark
/// @param robj   the current runnable
/// @param nset   the current nameset
/// @param quark  the quark to define as const
/// @param object the object to set
virtual Object* cdef (class Runnable* robj, class Nameset* nset,
const long quark, Object* object);

/// set an object to this object
/// @param robj   the current runnable
/// @param nset   the current nameset
/// @param object the object to set
virtual Object* vdef (class Runnable* robj, class Nameset* nset,
Object* object);

/// set an object to this object by quark
/// @param robj   the current runnable
/// @param nset   the current nameset
/// @param quark  the quark to set this object
/// @param object the object to set
virtual Object* vdef (class Runnable* robj, class Nameset* nset,
const long quark, Object* object);

/// evaluate an object in the current nameset
/// @param robj   the current runnable
/// @param nset   the current nameset
virtual Object* eval (class Runnable* robj, class Nameset* nset);

/// evaluate an object in the current nameset by quark
/// @param robj   the current runnable
/// @param nset   the current nameset
/// @param quark  the quark to evaluate in this object
virtual Object* eval (class Runnable* robj, class Nameset* nset,
```

```
  const long quark);

    /// apply an object with a set of arguments
    /// @param robj   the current runnable
    /// @param nset   the current nameset
    /// @param args   the arguments to apply
    virtual Object* apply (class Runnable* robj, class Nameset* nset,
  class Cons* args);

    /// apply an object by quark with a set of arguments
    /// @param robj   the current runnable
    /// @param nset   the current nameset
    /// @param quark  the quark to apply this arguments
    /// @param args   the arguments to apply
    virtual Object* apply (class Runnable* robj, class Nameset* nset,
  const long quark, class Cons* args);

    /// apply an object by object with a set of arguments
    /// @param robj   the current runnable
    /// @param nset   the current nameset
    /// @param object the object to apply this arguments
    /// @param args   the arguments to apply
    virtual Object* apply (class Runnable* robj, class Nameset* nset,
  Object* object, class Cons* args);

    /// apply an object with a vector of arguments by quark
    /// @param robj   the current runnable
    /// @param nset   the current nameset
    /// @param quark  the quark to apply these arguments
    /// @param argv   the vector arguments to apply
    virtual Object* apply (class Runnable* robj, class Nameset* nset,
  const long quark, class Vector* argv);

  public:
    // the memory allocation
    void* operator new      (const t_size size);
    void* operator new   [] (const t_size size);
    void  operator delete    (void* handle);
    void  operator delete [] (void* handle);
  };
}

#endif
```

INDEX

Colophon

This manual was written for the LaTeX documentation preparation system. A custom document class was designed by the author. The document style has been simplified as to produce a high quality technical manual. Title, chapter and section names have been produced with an Helvetica font. The document has been produced with a 10 points Times font. Both fonts are assumed to be in the public domain. The documentation is available in both A4 and letter format.