s11n
an Object Serialization Framework for C++

Version 0.8.x
stephan@s11n.net - http://s11n.net

15th May 2004

Abstract

CVS version info: $Revision: 1.11 $
Maintainer: stephan@s11n.net

This document describes s11n and "s11nlite", an object serialization framework for C++. It serves as a supplement to the s11n API documentation and source code, and is not a standalone treatment of the entire s11n library. Much of this documentation can be considered "required reading" for those wanting to understand s11n's features, especially it's advanced ones.

s11nlite, introduced in s11n version 0.7.0, simplifies the s11n interface, providing the features that "most clients need" for saving and loading arbitrary objects. It also provides a reference implementation for implementing similar client-side interfaces. The author will go so far as to suggest, with uncharacteristic non-humbleness, that s11nlite's interface ushers in the *easiest-to-use*, *least client-intrusive*, *most flexible* general-purpose object serialization library ever created for C++.

Users who wish to understand s11n are strongly encouraged to learn s11nlite before looking into the rest of the library, as they will then be in a good position to understand the underlying architecture and framework, which is significantly more abstract and detailed than s11nlite lets on. Users who think they know everything about serialization, class templates and classloaders are *still* encouraged to *give s11nlite a try*: they might just find that it's just too easy to *not* use!

Contents

1 Preliminaries

ACHTUNG: this is a live document covering an in-development software library. Ergo... it may very well contain some misleading or blatantly incorrect information!

1.1 License

The library described herein, and this documentation, are released into the Public Domain. Some exceptional library code falls under other licenses such as LGPL, BSD, or MIT-style as described in the README file and their source files.

All source code in this project has been custom-implemented or uses sources/classes/libraries which fall under LGPL, BSD, or other relatively non-restrictive licenses. It contains no GPL code, despite it's "logical inheritance" from the GPL'd libFunUtil. Source files which do not fall into the Public Domain are prominently marked as such.

To be perfectly honest, i prefer, instead of *Public Domain*, the phrase *Do As You Damned Well Please*. That's exactly how i feel about sharing source code.

1.2 Disclaimers

1. This manual will make *no sense whatsoever* to most people. It is target at experienced C++ programmers ("intermediate level"+), and makes many assumptions about prior C++ knowledge.
2. *Don't let the size of this manual make you think that using s11n is difficult!* Using s11n (especially s11nlite) is simple and straightforward, even for non-guru C++ coders. It also has a number of "power user" features which can be exploited by those who truly understand the architecture.
3. s11n is, in code terms, still very much under development. The basic model it is based on, however, has proven to be inordinately effective and low-maintenance since it was introduced in the QUB project (qub.sourceforge.net) by Rusty "Bozo" Ballinger over 3 years ago. This implementation refines that model, vastly expanding it's capabilities.
4. This library is PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
5. Reading disclaimers makes you go blind. ;)
6. Writing them is even worse. :/

And, finally:

This library is developed in my private time and the domain and web site (e.g.) are funded by myself. With that in mind: unless i am kept employed, this project may "blink out" at any time. That said, this particular project holds a special place in my heart (obviously, or you wouldn't be seeing this manual and all this code), so it often does get a somewhat higher priority than, e.g., dinner or lunch.

1.3 Feedback

By all means, please feel free to submit feedback on this manual and the library: positive, negative, whatever... as long as it's constructive it is always happily received. While most development-related communication happens via private emails, we do have a public mailing list where anyone may post their thoughts:

s11n-devel@lists.sourceforge.net

If this gives you any idea of how seriously feedback is taken:

- The whole 0.7.0 rewrite, and the abstractions and simplifications which grew out of it, were triggered by **Ton Oguara**'s feedback about his problems serializing class templates. (That is indeed a deceptively tricky problem, and the older code could only handle non-trivial cases with a non-trivial amount of code generation. The 0.7x framework can do this with "relative" ease.)
- This particular document (the one you're reading now), was largely inspired by **Gary Boone**'s feedback on the difficulties of getting started with s11n. Also, the changes in the registration processes from 0.7x to 0.8 were inspired by Gary.

In a very real way, s11nlite was developed because these two gentlemen took some time to share their thoughts.

The contact address, should you also feel compelled to write what you *really* think about s11n, is at the top of this document.

Now, i can't promise to rewrite everything every time someone wants a change, but all input is certainly considered. :)

Whatever it is you're trying to save, s11n *wants* to help you save it, and goes through great pains to do some deceptively difficult tricks to simplify this process as much as practically possible. If it *can't* do so for your use-cases, then please consider helping us change s11n to make it capable of doing what you'd like it to. It is my firm belief that the core s11n framework can, with very little modification, save *anything*. What is currently missing are the algorithms and containers which may further simplify the whole process, but only usage and experimentation will reveal what that toolkit needs to look like. If you come across some great ideas, please share them with us!

:)

— stephan

1.4 Credits

Very briefly, in no particular order:

- My parents, Bonnie Pickartz and Joseph Hudgins, because they just need to be thanked in general.
- Rusty "Bozo" Ballinger wrote the conceptual forefather of s11n (http://libfunutil.sourceforge.net).
- Ton Oguara accidentally inspired the whole 0.6 -> 0.7 rewrite/refactor.
- Gary Boone provided valuable feedback on a range of documentation and features, particularly on making it easier for developers to get started with s11n. Many of the 0.8.x improvements exist because of Gary's feedback.
- Marshall Cline, of C++ FAQ fame, has provided a number of corrections, insights and suggestions regarding the handling of cyclic graphs.
- Roger Leigh provided the information needed to add `libltdl` support to the classloader.
- Tom, from comp.lang.c++, provided an interesting fix for an "interface annoyance" in the classloader.
- "Ashran", my office-mate, business partner and best friend, often puts up with me ranting (like a madman, i might add) about the latest s11n/classloader breakthroughs. He puts up with a lot more than that, actually[1]. ;)
- Peter "What's Happnin!!?!!?!?!" Angerani, my long-time friend and mentor, for his continued support and feedback.

Various published authors have, rather unknowingly, had *profound* impacts on various design decisions during s11n's evolution:

[1] WHADYA *MEAN* you didn't pay the office rent yesterday!?!?

- Scott Meyers - a *huge* percentage of my code is influenced by Scott's always-practical advice. Here's your biggest fan, Scott!
- Andrei Alexandrescu[2] - his *Modern C++ Design* was the necessary catalyst i needed for realising the classloader implementation, and provided the basis for the internals of the `s11n::phoenix<>` class, which is used *extensively* by s11n.
- Herb Sutter - A couple of his (very numerous) articles have led to direct changes in this library. e.g., a breaking-down of some of the classloader's member interface into free functions was inspired by his "What's in a class?" article.
- Stephen Dewhurst, author of *C++ Gotchas*: every time i write "template class" and correct it to "class template", or change the word "method" to "function", i think of Stephen. ;)

2 Introduction

So you want to save some objects? Strings and PODs[3]? Arbitrary objects you've written? A `std::map<int,std::string>` or `std::list<MyType *>`?

What?!?! You've got a `std::map< std::list<int *>, MySerializable<X *> * >`[4]?!?!?

No problem:

s11n is here to ***Save Your Data***, man!

Historically speaking, saving and loading data structures, even relatively simple ones, is a deceptively thorny problem in a language like C++, and many coders have spent a great deal of time writing code to serialize and deserialize (i.e., save and load) their data. The s11n framework aims (rather ambitiously) to completely end those days of drudgery.

s11n, a short form of the word "serialization"[5], is a library for serializing... well, just about any data stucture which can be coded up in C++. It uses modern C++ techniques, unavailable only a few years ago, to provide a *flexible*, fairly *non-intrusive*, *maintenance-light*, and *modern* serialization framework... *for a programming language which sorely needs one!* s11n is particularly well-suited to projects where data is structured as hierarchies or containers of objects and/or PODs, and provides *unprecedentedly simple* save/load features for most STL-style containers, pretty much regardless of their stored types (section 7.4).

In practice, s11n has far exceeded it's original expectations, requirements and goals, and it is hoped that more and more C++ users can find relief from Serialization Hell right at home in C++... via s11n.

A brief history of the project and a description of it's main goals are available at:

http://s11n.net/history.php

2.1 Scope of this document

This document *does not* cover every detail of how s11n works (that'd take a whole book[6]). It *does* tell clients what they need to quickly get started with s11nlite (and, by extension, s11n). For complete details you'll need this document, the API docs, *and* the source code. That said - i try to get all the client-necessary text into this document.

As always, the sources are the definitive place for information: see the README for the locations of the relevant files.

2.2 WTF is s11nlite?

s11nlite is a "light-weight" s11n sub-interface written on top of the s11n core and distributed with it. It provides "what most clients need for serialization" while hiding many of the details of the "raw" core library from the client (trust me - you *want* this!). Overall it is *significantly* simpler to use but, as it is 100% compatible with the core, still has access to the full power "under the hood" if needed. s11nlite also offers a potential starting point for clients wishing to implement their own serialization interfaces on top of the s11n core. Such an approach can free most of a project's code from direct dependencies s11n by hiding serialization behind an interface which is more suitable to the

[2] i can even spell it now without looking it up ;).

[3] Plain Old Data - int, char, bool, double, etc.

[4] The only (remaining) inherently difficult part for this one is getting the proper type *names* for each component of the container heirarchy! This problem discussed at length in this documentation, the s11n sources, and the class_loader library manual. It's not as straightforward as it may seem...

[5] This term was coined by Rusty Ballinger in mid-2003, as far as i am aware. It follows the tradition set by "i18n", which is short for "internationalization".

[6] But i'd be happy to entertain a publishing offer! :)

project. (Such extensions are beyond the scope of the document, but feel free to contact the development list if you're interested in such an option.)

Users new to s11n are *strongly* encouraged to learn to use the code in the s11nlite namespace before looking into the rest of the library. Doing so will put the coder in a good position to understand the underlying s11n architecture later on. Users who think they know everything are still encouraged to give s11nlite a try: they might just find that it's just too easy to *not* use! Don't let the 'lite' in the name *s11nlite* fool you: it's only called s11n*lite* because it's a subset of an even more powerful, more abstracted layer, known as "the s11n core" or "core s11n." For those who just can't wait to dig in: see the README file for the code locations.

s11nlite is still very infantile - as of March 15, 2004, well over 60% of it's code-base is only 2 weeks old and some 80%+ of the library manual has been rewritten from scratch. That is, 40+ pages of *new* docs, plus well over *500k* of brand new source files(!!!), all under 16 days of age. Thus, there are *bound* to be bugs or oversights.

That said, the general model itself has proven to be *very* effective. Historically, this is the 3rd time the architecture been significantly refactored, and it is evolving to be more and more useful with each iteration. This particular iteration is light years ahead of it's predecessors, in terms of power and flexibility, and is also much simpler to work with and extend.

2.3 Main features

The library's primary features and points-of-interest are:

- Quite possibly the *most flexible* and *easiest-to-use* C++ serialization framework *in the known universe.*[7]
- Provides client code with *easy* de/serialization of arbitrary streamable types, user-defined Serializable types and various STL containers.
- Lends itself well to a large number of uses, from de/serializing arbitrary vectors or maps of data (a-la config files) to saving whole applications in one go.
- Does not tie clients to a specific Serializable interface/heirarchy. The internally-used interfaces can be *easily* directed to use client-specific interfaces, which need not even be virtual. This means that the library's interface can be made to conform to client-side objects' needs, as opposed to the other way around.
- Serializable Proxying allows clients to attach proxy classes to *ANY* given type, such that the proxy type is delegated all de/serialization operations. The end result is that it is possible to serialize a given type without having to touch a line of that type's code, nor does that type have to know it's playing along. Also, this allows swapping out serialization techniques whenever you like (in this model, that simply means changing the internal structure of some "Data Nodes" to suit your needs/preferences).
- Integration into existing class hierarchies is straightforward, quick, relatively painless and can often be incrementally applied to subsets of a project over time, as needed, as opposed to forcing a client to completely refactor. In fact, using proxies means client classes don't normally have to change *at all* to be transformed into "True Serializables."
- The data persistance model inherently does not suffer (as, e.g., Java's does) from the problem of invalidating serialized data every time an internal change is made to a Serializable data type. It's properties-based system ensures that legacy data do not become invalid until developers *want* them to become so.
- It sports *compile-time type-safe classloading*, even for those classes loaded via DLLs, without the use of a *single* type-cast (neither in the client nor in the library). The classloader can load just about *any* classes, including 3rd-party classes, without them knowing they ar participating. The classloader mini-framework has a huge number of uses and features outside the context of s11n... and it therefor has own web site: http://s11n.net/class_loader/
- The API is *100% data-format agnostic* and places *no file naming conventions* client data files. Several different data format handlers (aka, Serializers, 5(!) of them) are provided with the library, and adding custom Serializers is fairly painless: all you need is an input parser and an output formatter.
- Optional client-transparent zlib and bz2lib file de/compression, for 60-95% file size reduction.
- The i/o sub-framework is stream-centric, not file-centric. This sub-module is effectively optional: clients are not required to use *any* of the supplied i/o code, as long as they supply their own file parsers (lexers, for example).
- The primary data structures follow STL [Standard Template Library] conventions and are container/functor/algorithm-centric, thus many generic algorithms can be easily applied to them. The library comes with several useful functors and algorithms for working with serialized data.

[7] i'm willing to admit that i may be mistaken, but i have yet to see a contender which comes even *close*.

- Provides a framework under which code from many different projects can share data, even if they internally use different Serializable interfaces and data formats.
- Uses only standard C++ constructs, no compiler-specific extensions.

Okay, okay, i'll stop there! ;) (The list *really does* go on and on!)

2.4 Notable Caveats (IMPORTANT)

It would be dishonest (even if only mildly so ;) to say that s11n is a magic bullet - *the* solution to *all* object serialization needs. Here are the currently-known major caveats which must be understood by potential users, as these are type types of caveats which may prove to be deal-breakers for potential s11n users:

- s11n, at it's core, can be quite difficult to grasp. It's not the details which are difficult for most people, i think, but the fact that the details are lie beind very abstract "conventions" and "close approximations" instead of hard-and-fast rules and defintions. (This is largely a side-effect of my own personal philosophies, and i appologize if the core seems a bit... eccentric.)
- The supplied build tree will only run on GNU-based systems. That is, systems running all the common GNU tools like `gzip`, GNU `make`, `bash`, and other exceedingly common Open Source tools, like `perl`. That said, the code itself should be easily portable to other build systems, so long as those hosts support appropriate compilers (see below). We will gladly host build-related files for other platforms or build environments (e.g., GNU Autotools, Microsoft environments, etc.) in the distribution and/or web site, should users submit those.
- Requires a relatively recent, ISO-conformant C++ compiler with excellent support for class templates. Only known to work with GCC 3.2x and 3.3x, and known to *NOT* work with GCC 2.9x. Based purely on what i've read of Microsoft Visual C++, there is no hope of this code working on any version lower than 7.1 (i personally have very little experience with MSVC).
- Some very small pieces of code are specific to Unix-like systems (e.g., `dlopen()/ltdlopen()`). Users experienced with other platforms are encouraged to fix that :)! (Grep the tree for dlopen and you'll jump right to it.)
- s11n is untested with binary data. It "should be possible", but implementing it in terms of the current Serializers (e.g., as string-encoding conversions) would be *extremely* inefficient. That said, any data which can ultimately be represented as a `std::string` should pose no problems at all for s11n.
- As it is heavily based on class templates, it is implemented largely as inlined code in header files (for complex linking reasons). The end effect on clients is that compilation times and object/binary file sizes *do* suffer. Some code is implemented in implementation files, so clients must still link to the s11n library, either statically or dynamically, just as they would for any typical C/C++ library.
- Due partly to the side-effects of heavy reliance on class templates, s11n is probably unsuitable for systems with very limited filesystem space or main memory (e.g., embedded systems, handheld computers, etc.).
- s11n is untested in multi-threaded environments. See section 16.3 for more details/speculation.
- It is driven with Generic Programming and reusability/maintenance in mind, not High-performance Computing, and thus it may not be performant enough for projects which need, *really*, ***really fast*** code. (That said, s11n is acceptably fast for all uses i've had for it. Make your own judgement.)
- s11n is 100% driven by my hobbies and my coding needs, and is constantly under experimentation. Thus it is subject to change radically *at any given moment* (as it did from 0.6.x to 0.7.x). Then again, it's also potentially subject to sitting still for some significant stretch of time.

2.5 Compatibility with earlier s11n versions

(This is only of interest for clients who have code based on 0.6.x and earlier. That's *probably* only me.)

As of version 0.7.0, the 0.6.x interface ("old-style", as it is now known) is "officially deprecated." That means it's out-moded and should be avoided by future client code. The new interface, especially s11nlite, is *highly* preferred.

The 0.6.x-based interfaces are no longer, as of 0.7.1, shipped in the source releases. The remaining copies exist on the s11n download site and as relics in a CVS server, but the code is of little use, as it has been completely supplanted by the new core.

With 0.8.0 a *lot* also changed - most of the 0.7.x concepts are still valid, but some usages have changed.

3 Core concepts

Users of s11n should read this section carefully - it details the major components and terms of the architecture, which will make understanding the library much simpler.

3.1 Terms and Definitions

Below is a list of core terms used in this library. The bolded words within the definitions highlight other important terms defined in this list, or denote particularly significant data types. This bolding is intended to help reinforce understanding of the relationships between the various elements of the s11n library.

- **s11n** - several meanings:
  - A short-hand form of the word "serialization", following the tradition set by the word "i18n" as a shorthand for "internationalization." That is - by replacing all but the left- and right-most letters of the word with the *number* of letters replaced.
  - The name of this library.
  - Serialization as a computing domain.
  - Other, more context-specific, meanings.
- **Data Node** - a generic term for map-like types which store arbitrary key/value properties and child nodes, plus some meta-data (like type information for the stored data, if needed). They are structured in a tree-like fashion, DOM-style. In s11nlite this role is played by the `s11n::data_node` type, though core s11n supports any types which conform to the conventions laid out by that type. (The core doesn't actually know that type exists.)
- **serializable** (with a *small* "s")- the property of being able to be save and restore state, e.g., to allow persistant states across application sessions, network connections, etc.
- **Serializable Type** or **Serializable** (with a *big* "S") - any type for which s11n recognizes a **Serializable Interface**, either implemented directly by a Serializable type or via a **Serialization Proxy**. **Serializables** save their state in **Data Nodes** during **serialization** and restore their state from **Data Nodes** during **deserialization**.
- **Serializable Proxy** or **Serialization Proxy** - a functor (or possibly two) which registers with s11n as being the handler for de/serialization of a given type. By extension, the proxied type is considered to be a full-fledged **Serializable**. All **de/serialize operations** s11n performs on behalf of that type are delegated to the proxy type. This allows, amongst other things, transparent serialization of 3rd-party classes.
  Proxies are *not* technically **Serializables** - they are, more properly, the implementation for a **Serializable**'s **serialization operators**. (Got that?)
- **serialization, to serialize** - several meanings:
  - To save the state of a **Serializable** into a **Data Node**.
  - To save a **Data Node** to a data stream. This emits it to a **Serializer**-specific grammar.
  - Several other context-specific meanings.
- **deserialization, to deserialize** - the converse of **serialize**:
  - To restore the state of a **Serializable**, presumably using data from a **Data Node**.
  - To load a **Data Node** from an input stream.
- **de/serialization** or **de/serialize** - shorthand forms of "deserialization and serialization" and "deserialize and serialize."
- **Serializer** - a type responsible for converting data nodes to and from a specific grammar. For example, some Serializers use a XML dialect, while others use custom formats. Theoretically, any data which can be structured in a DOM-like fashion (even if only via structural transformation) can be handled by Serializers. In s11n *Serializers* are also always *Deserializers* (at least logically, in terms of the interface), with the one minor exception that **Serializable Proxies** may be implemented in terms of two separate functors[8].
- **serialization operators**, **de/serialize()**, or **Serializable Interface** - a generic name for the pair of de/serialize functions which **Serializables** and **Serializable Proxies** have, regardless of the actual names or argument types of the functions. Sometimes also used to refer to the de/serialize functions within other interfaces, such as the s11n core.

[8] This is, unfortunately, occasionally necessary to avoid potential ambiguity when calling their serialization operators.

- **de/serialization operations** - In abstract terms: generic terms encompassing any functions which trigger a chain of events which lead through the s11n de/serialization core (and, presumably, back). In plain English: `s11n::de/serialize<>()`, and related functions, fall into this category. In very specific terms, it refers to any class which end up forwarding through the `s11n_api_marshaler<>` interface.

- **Default Serializable Interface** - Serializables which implement both of their serialization operators as `operator()` are said to follow the *Default Serializable Interface*. Types which do this do not need to tell s11n what their serialization interface looks like - it will pick them up automatically.

- **Load/Save vs De/Serialize** - By s11n convention, the words "save" and "load" are used when dealing with streams or files, and "serialize" and "deserialize" are used when refering to saving or restoring the state of a **Serializable** to or from a **Data Node**. Sometimes the words are used interchangeably and, while it is technically correct in many cases, such usage is considered "marginally ambiguous" in s11n. That said, we aren't terribly pedantic about this point ;).

- **Classloader** - an object used to search for classes based on a lookup key. In s11n this lookup key is conventionally the string form of a class' name. Classloaders are used during deserialization to load the proper type for a given node (this is necessary in order to support polymorphic serialization). The s11n classloader has support for loading classes from DLLs, but that feature is not covered much in these docs. Classloaders work primarily not off of specific "concrete" types, but off of Base Types, as described briefly below. See http://s11n.net/class_loader/ for more detail than you probably want to know about these.

- **T's classloader**, or **the T classloader** - Refers the the classloader which uses type T as it's point of reference for registering and loading classes. More specifically, it means `s11n::class_loader<T>`, a classloader which supports loading T types. Subtypes of T should be registered with T's classloader, regardless of whether or not they also register with their own classloader (e.g., `class_loader<SomeTSubType>`).

- **Base Type** - in s11n, especially in the context of a **classloader**, this is used to mean the base-most type which a given **classloader** "knows about." This type is used for registering subtypes of Base Types with the Base Type's **Serializable Interface**, and is a crucial detail for classloading purposes. It is covered in massive detiail in the classloader's library manual. In a broader sense, Base Types are used as contexts for marshaling the s11n and client-side **Serializable Interfaces** into internally-compatible forms. Base Types are often used as contexts, in the form of template parameters for functions and classes for which this Base Type distinction is significant (e.g., those dealing with de/serialization and classloading)[9].

- **Streamable [Types]** - In the context of s11n this means any type for which istream>> and ostream<< operators can be applied to successfully save/restore the state of an object of that type. This inherently includes all PODs, `std::string`, and any client-supplied types which meet these conditions. This also implicitely *excludes* all pointer types. Note that **Serializables** are not implicitely Streamable, as s11n does not actually deal with streams at it's core.

- **SAM, the s11n API Marshaler** - SAM is the layer of s11n responsible for acting as a communication channel between s11n's internal API and any client-side APIs, including, but not at all limited to, forwarding requests to **Serializable Proxies**. SAM allows clients to transparently proxy the s11n interfaces, is covered in section 13.

- **core s11n** or **the s11n core/kernel** - These are generic terms referring to the core-most functions in s11n. Specifically, this is limited to the classloader-related functions and de/serialize() variants defined in data_node_serialize.h. Everything else, from the **Serializers** to the s11nlite interface, is build around this core.

Did you get all that?

Using the library is not as complex as the above list may imply, as the rest of this documentation will attempt to convince you. Yes, the details of serialization and classloading, especially in a lower-level language like C++, are *downright scary*. s11n tries to move the client as far away as possible from those scary details, and it goes to great pains to do so. However, some understanding of the above terms, and their inter-relationships, is critical to making *full* use of s11n (it goes well beyond what this manual covers).

Some non-s11n-related terms show up often enough in this documentation that readers not familiar with them will be at a disadvantage in understanding the library. Briefly, they are:

- **i.e.** - "in other words" or "in effect" (from the Latin *id est*[10])

- **e.g.** - "for example" or "example given" (from the Latin *exempli gratia*)

[9] Normally such template parameters are named SerializableType or NodeType.

[10] Source: http://www.wsu.edu:8080/~brians/errors/e.g.html

- **Algorithm** - we use the same general meaning as in common STL usage: a computuation, normally one commonly used in numerous contexts, which is genericized in form such that it can be applied to a wide range of types which meet the published set of conventions for that algorithm. Like **functors**, understanding algorithms is essential to effectively using the STL and, by extension, STL-based libraries such as s11n, and algorithms/functors often go hand-in-hand.
  For *numerous* well-published examples see those in the STL itself, defined in the ISO-standard `<algorithm>` header file. Most of the algorithms for s11n are in lib/standalone/src/algo.h (generic algos) and lib/node/src/data_node_al (s11n-specific).

- **Functor** - a function or a struct/class type implementing function-call semantics. i.e., a type implementing one or more `operator()` member functions. Functors are a cornerstone of all STL-style development, and must be well-understood before one can make full use of s11n. Most of the client-usable s11n-related functor types are declared in lib/standalone/src/functor.h (generic functors) and lib/node/src/data_node_functor.h (s11n-specific).

- **ODR, the One Definition Rule** - C/C++'s rule which, put simply, basically states that no function or type may be *defined* (i.e., implemented) more than one time in any given binary/library. (This is not an arbitrary rule, but a technological limitation, akin to std::map being able to only have one object with any given key. In any case, it's rather a *sane* behaviour, if you ask me.)
  In s11n ODR is an oft-heard term because it's template-based nature, in particular it's use of macros and header files to generate "behind-the-scenes" utility and marshaler class template specializations at complile-time, makes it quite succeptible to ODR violations if some simple, non-obstructive rules are not followed (as described elsewhere in this manual). (Trust me, once you realize how it works this is never a practical hinderance, and it is trivial to avoid once you see it happen it a few times and understand it's nature.) With the release of version 0.8.0, the most commonly-occurring ODR-related problems are believed to be solved.

- **Style Points (SP)** - an abstract, often poorly-understood and underestimated, unit of measurement of "how much *Style*" a particular piece of code exhibits. Poorly-designed code gets minus points, whereas especially clever code may get plus points (or may, as is occasionally the case, actually be too clever for it's own good, and get no points at all). The measurement system for Style Points is not standardized. One common way for one developer to communicating that s/he wishes to assign SP to, or substract SP from, another developer is to say say something like, "+1", or "-1". A phrase like, "cool code!" implicitely carries at least one SP, whereas the phrase, "great hack!" or "you rock!" is generally worth several SP (at least from the receiver's perspective).
  It is significant to keep in mind that SP declared by non-developers simply go to `/dev/null` - they neither count nor discount the recipient, except possibly in his or her own ego[11]. Additionally, the amount of SP a given reward or pentalty gives or takes may be adjusted by the relative experience levels or reputions of the giver and receiver. e.g., a 6-month C++ newbie giving +1 SP to a 10-year veteran is not worth *nearly* as much the other way around.
  The giving of Style Points is sometimes referred to as "schenking" (past tense: *schenked* or *schenkt*), derived from the German verb *schenken*, meaning "to give [free of cost/as a gift]."
  As software developers mature[12] they invariably begin, at some indefinate point, to concentrate on Style as much as they do on the algorithms they develop. This is a natural part of a developer's growth as a professional, just as it is in any field, and thus experienced coders can general "pick up SP" much more readily than greenhorns can.
  s11n trivia: To date (17 March 2004), s11n has officially gotten several "neat idea" mails (0 or 1 SP each), one "(really cool!) hack!" mail (probably 2-3 SP) one "you rock!" mail (i'll also put that at 2-3 SP), and one "work of art!" mail (regarding the docs, but it doesn't count: it came from my mother, who's not a developer ;). And i've loved reading every one of them. :)

3.2 The Official *Grossly Oversimplified Overview* of the s11n architecture

s11n is built out of several quasi-independent sub-modules. "Quasi-independent" meaning that they mostly rely on *conventions* developed within other modules, but not necessarily on the exact *types* used by those modules. Such design techniques are a cornerstone of templates-based development, and will be a well-understood principal to STL coders, thus we won't even begin to touch on it's benefits, uses, and multitudinous implications here.

Shameless Plug[13]*:*

> This particular aspect of s11n's design is critical to s11n's flexibility, and is one of the implementation details which catapults it *far* ahead of traditional serialization libraries. It is, for example (and as far

[11] And we programmers, by and large, have a repution for living the majority of our lives in exactly that space. ;)

[12] As developers, not necessarily as human beings.

[13] Such a plug is typically worth approximately *-1 Style Point*, a cost from which this plug is not excempt. In fact, these docs have so many shameless plugs and outbursts of jubileaum that i'll go ahead and dock the document as a whole -10 SP. ;)
(i wouldn't be preaching it if i didn't believe honestly it, though, so the devotion's gotta be worth a couple of SP!)
What is a Style Point? See section 3.1.

as i am aware), the first software of it's kind which allows client libraries to transparently adapt the framework's interfaces to the client's interface(s), and to transparently adapt *other clients'* Serializable interfaces (and, additional, transparently adapt to *them*). In most (all?) other libraries this model is the other way around: the client has to do all adapting himself. Consider, e.g., that *any* type can converted to a Serializable without, e.g., subclassing anything at all. That is, a client can have 1047 different classes - each with their own serialization interfaces - and they can all transparently de/serialize each other *as if they all had the same function-level interface*[14].

Enough plugging. Let's briefly go over s11n's major components, in no particular order:

- **Classloader** - a factory for creating classes based on lookup keys (e.g., class name). This is a critical element for proper polymorphic deserialization, particularly when loading classes on-the-fly from external sources (e.g., a DLL).
  As of version 0.7.2, the classloading framework has been simplified via the cllite namespace, which is the classloader's equivalent of s11nlite.

- **data_node** - this is the default/reference implementation for Data Note types. It is supported by a number of s11n-supplied algorithms and functors, though it has no direct dependencies on them. It is significant for this library that data_node has NO dependencies on any other s11n-related code, and should never develop such dependencies: it is a standalone reference implementation of s11n's **Data Node** concept.

- **Serializers** - these objects are responsible for marshaling Data Nodes to and from specific file formats (grammars). The library currently s11n ships with 5 Serializers, but s11nlite only uses one of them by default (*which one* it uses is not strictly defined by the interface), and can be changed at runtime by calling s11nlite::serializer_class().

- **Core de/serialize() functions** - a set of functions which hide the API marshaling that goes on for translating arbitrary Serializable interfaces into something each other can use. At the application level, these functions typically make up the heart of the client-side s11n interface, whereas at the library- and class- levels the available functors and algorithms a much more likely to play a heavy role.

- **Serialization API marshalers** (SAM)- the core de/serialize passes all of their de/serialize request through these. These types can be swapped out transparently, customizing the serialization interface on a per-base-type instance. This feature is used, for example, to direct certain types to use Serializable Proxies, or to implement pointer-to-reference type translation as needed. These marshaler filter every single de/serialize call made via the core, and thus the ability to replace them on the client side gives client code 100% plug-in access to the framework's de/serialization core, without having to know the details of how everything is marshaled. **SAM** is covered in section 13.

- **s11nlite** - a subset of the above layers, wrapped up in a tidy little interface providing for most common client object serialization needs. Intended also as a sample client-side interface implementation. That is, by implementing something like s11nlite a project can completely hide it's objects from *any* knowledge of libs11n, helping to support the "non-intrusion principal" which s11n tries to uphold.

- Generic helper functors and algorithms to support internal and client-side manipulation of Data Nodes and Serializers, also helpful for s11nlite.

There are also a number of less-visible support layers/classes/functions. See the README file for an overview of where each part of the library lives in the source tree. The API docs reveal the whole spectrum of available objects (many of which are internal or special-case, and can be ignored by clients).

Some of the sub-sub layers exist purely as code generated by macros (such as the classloader registration macros), e.g. to install client-specific preferences into the library.

3.3 Process Overview

3.3.1 Serialization

In the abstract, this is normally what happens for a serialization operation:

1. Client requests the serialization of a Serializable. This is initialized by passing the Serializable into a data container (e.g., s11n::data_node) via the s11n serialization interace. e.g. s11nlite::serialize<MySerializable>(mynode,myseriali

2. s11n proxies the request to the registered (or default) interface and passes the target node and source Serializable to the registered interface.

[14] Whereas they do all implicitely share a common *logical* intereface - that of a Serializable, as defined by s11n's conventions.

3. The serialize operator's implementation should save the Serializable's state into the data node. It returns true on success and false on error.
4. s11n returns a data node to the client, presumably populated with the data from the Serializable.
5. Client selects a Serializer type and sends the node to it, along with a destination stream/file.
6. Serializer formats the node into the Serializers grammar.
7. The client gets notification of success or failure (true or false, respectively).

Recursive serialization can be triggered, e.g., in a serializable operator's implementation, which serializes a child Serializable or a container of Serializables.

Note that in s11nlite the Serializer selection steps are abstracted away to simplify the interface.

3.3.2 Deserialization

While there are at least two client-side approaches to deserialization, most requests normally go more or less like this:

1. Client requests the deserialization of a Serializable Type from a data stream/file. e.g., s11nlite::deserialize<MySerializable>
2. s11n analyses the stream to find a matching Serializer class, then passes the stream off to the that class.
3. Serializer parses the stream into a tree of data nodes and returns the root node to s11n.
4. If the client requested the loading of a Data Node, as opposed to a Serializable, then the root node is passed to the user and processing stops here.
5. s11n looks at the root node's impl_class() member and uses the Serializable Type's Classloader to load a concrete Serializable implementation named after the impl_class(). If it fails to find a class, processing stops.
6. s11n marshals the data-to-be-deserialized to the registered deserialization interface for Serializable's type.
7. Deserialize operator's implementation should restore the Serializable's state from the source data node. If it returns false processing stops.
8. s11n destroys the now-unnecessary data node.
9. s11n returns a (MySerializable *) to the client, which the client now owns.

The interface also supports deserializing nodes directly into arbitrary Serializables, effectively bypassing steps 2-5.

3.4 Node Names and Property Key naming conventions (IMPORTANT!)

When saving data each node is given a name, fetchable via the `name()` function. Node names can be thought of as property keys, with the node's content representing the value of that key. Unlike property keys, node names need not be unique within any given data tree. All nodes have a default name, but the default name is not defined (i.e., clients can safely rely on new nodes have *some* Serializer-parseable name).

In terms of the core s11n framework, the key/node names client code uses are irrelevant, but most data formats will require that they follow conventional variable name syntax:

> ***alphanumeric and underscores only, starting with a letter or underscore.***

Any other keys or node names will almost certainly *not be loadable* (they will probably be saveable, but the data will be effectively corrupted). More precisely, this depends on the data format you've chosen (some don't care so much about this detail).

Numeric *property keys* are another topic altogether. Strictly speaking, they are not portable to all parsers. More specifically, numeric keys (even floating-point) are handled by the parsers supplied with this library (even the XML ones), but the data won't be portable to more standards-compliant parsers. Thus, if data portability is a concern, avoid numeric keys altogether, and also be aware of the algorithms which uses "dummy" numeric keys when storing containers of objects, which is sometimes necessary to keep the containers' original ordering.

Serializable classes normally do not need to deal with a node's `name()` except to de/serialize child Serializables. There are many cases where client code needs to set a node name manually, but these should become clear to the coder as they arise.

3.5 Overview of things to understand about s11n

After reading over the basic library conventions, users should read through the following to get an overview of what topics which should be understood by by clients in order to effectively use the s11n framework. Much of it is over-simplified here - this is an overview, after all. Additionally, some of it is true for s11nlite, but only partially true for core s11n.

- **s11n::data_node** is the basic type used to store arbitrary key/value pairs and child objects. It follows a DOM-style interface, so it's usage should be fairly straightforward. The core library actually supports any generic Data Node type which supports the same interface as data_node.
- The entire client-side interface for loading and saving all objects is declared in **s11nlite.h** and in namespace **s11nlite**. The core code is available in namespace s11n, and several of those functors and algorithms are useful within the context of s11nlite. That said, clients may directly use the core s11n, bypassing s11nlite completely, but learning s11nlite first is recommended.
- s11n is very container/functor/algorithm based, so it's usage should be familiar to experienced C++ users (especially users of the STL).
- s11n does not enforce a specific Serializable interface, but inherently supports the so-called *Default Serializable Interface*. Client-side classes which implement the default Serializable interface (described later) need no special registration as being Serializable types. Custom interfaces and proxies are *easy* to install, as described later.
- s11n's core is not stream-oriented, but container-oriented. That is, you serialize data to containers, and those containers get formatted to (or from) streams by Serializers. Thus s11n doesn't really care about file formats - it's core interface is 100% data format agnostic. For saving, clients must declare a format, but loading is dispatched to the appropriate parser depending on the content of the stream. That said, s11nlite uses a default Serializer type, so clients who don't care about the underlying data format need never worry about this detail (tip: see s11nlite::serializer_class()).
- Classloaders and their "BaseType" types are important concepts to understand in s11n, mainly for template-types reasons. They are covered *in detail* in the class_loader library manual, and will be explained a bit later on. *All* types which are to be *deserializable* must be registered with an "appropriate classloader." What that *really* means, in all it's technical glory, could easily turn into whole document! Be assured that this doc will try to tell you what you need to know in order to register your classes (it is 100% non-intrusive on classes). The hope is that most s11n use-cases won't require much client-side understanding of the subtleties of the classloader framework.

3.6 Notes on error/success values (i.e., justifying the bool)

s11n uses, almost exclusively, bool values to report success or failure for de/serialize operations. The reasons that bool was chosen are long, but here's a summary:

- It is my opinion that *some* error value is needed. Integer values must either be mapped to a known set of error codes - e.g. an enum - or be interpretted client-dependently. Neither of those approaches are terribly suitable for s11n, largely due to it's inherently abstract and generic nature.
- Based on usage history, i felt it was unnecessary to employ exceptions as the standard means of error reporting. (i partially regret this, but still generally feel that imposing exception conventions on the clients would not be a good idea.)
- If we consider the standard ostream<< and >>istream operators for a moment: yes, it is technically possible to check for an error after an extraction/insertion by checking the stream's state, but in practice this is rarely done, in particular for ostreams. Thus, i/ostream error checking conventions are oddly similar to s11n's, probably due to their logically similar roles as i/o marshalers.
- Related to the previous point: s11n's core is container-based, and how many coders check for proper insertion after a push_back() or insert()? None, because those operations (perhaps only by convention?) simply do not fail.
- i actually knew a coder once who (in Java) chose to return the String "success" to indicate success and non-"success" to indicate failure. i figure that's also not appropriate for s11n. ;)

s11n's conceptual ancestor, Rusty Ballinger's libFunUtil, uses void returns for it's de/serialize operations, which means that clients essentially can't know if a de/serialize fails. When designing s11n i strongly felt that clients need at least add *some* basic level of error detection, and finally settled on plain old booleans. There is in fact a comic irony in that decision: it is so rare that a de/serialization fails, that a void return type would do just as well for 98% of use-cases!

The seeming shortage of de/serialization failures can primarily be attributed to the following:

- The majority of the client-side part of s11n does not deal with i/o streams (in particular, with files).
- The points at which Serializables are given data nodes are far away (in interface terms) from the stream operations. Stream operations are, *by far*, the most likely point of failure in a serialization library (bad input format, file does not exist, out of disk space, write access fails, NFS connection cut, blah blah blah blah).
- The s11n core is container-based, and container insertions, as a general rule, do not fail. Also, container searches only fail in the sense that the data being searched for simply isn't there.
- In the event of a input stream/parse failure while reading in nodes the process will fail early enough that no deserialize operators are be called, so they can't very well fail, can they?

[... much later ...]

While returning a bool for a single de/serialization operation still seems reasonable, the logic behind it rather breaks down when a tree of objects is serialized. If any given object returns false the the serialization *as a whole* will fail. This implies that whole trees can be spoiled by one bad apple (no pun intended). In a best-case scenario only one branch of the tree would be invalidated, but... *is that a good thing*, to have partial data saved/loaded and have it flagged as a success? Of course not, thus s11n must generally consider one serialization failure in a chain of calls to be a *total* failure. This is it's general policy, though client/helper code is not required by s11n to enforce such a convention[15].

Furthermore, some specific operations, such as using for_each() to serialize a list of Serializables, may [will] have unpredictable results in the face of a serialization failure. Consider: in that case there is no reasonable way to know which child failed serialization, as for_each() will return the overall result of the operation. If the functor performing the serialization continues after the first error it will produce much different (but not necessarily more valid) results than if it rejects all requests after a serialization failure. The `data_node_child_deserializer<>` class , for example, refuses to serialize further children after the first failure, but this is purely that class' convention, not a rule. (In fact, that class has a "tolerant" flag to disable this pedantic behaviour.)

Ah... there is not 100% satisfying solution, and bools seem to meet the middle ground fairly well.

4 Serializable Interfaces: overview and conventions

Rather than overload you with the details of this right up front, we're going to *grossly oversimplify* here and tell you that the following is *the* interface which s11n expects from your Serializable types.

Each Serializable type must implement the following two methods:

A **serialize operator**:

> [virtual] bool operator()(*NodeType* & dest) const;

A **deserialize operator**:

> [virtual] bool operator()(const *NodeType* & src);

It is important to remember that *NodeType* is actually an abstract description: any type meeting s11n's Data Node conventions will do. s11nlite uses, unsurprisingly, s11n::data_node as the NodeType.

The astute reader may have noticed that the above two functions have the same signature... *almost*. Their constness is different, and C++ is smart enough to differentiate based on that. The s11n interface is designed such that it is very difficult for clients to have an environment where such ambiguity is possible.

These operators need not be virtual, but they may be so. Serializer proxy functors, in particular, are known for having non-virtual serialization operators, as are, of course, monomorphic Serializable types.

The truth is that s11n only requires that the argument be a compatible data node type and that the constness matches. s11n's core doesn't care what function it calls, as long as you tell it which one to use - how to tell s11n that is explained in section 9.

> **s11n trivia:** When the de/serialize operators are implemented in terms of `operator()`, a type is said to conform to the *Default Serializable Interface*.

[15] Especially when s11n's author can't even decide if s11n currently does The Right[est] Thing ;). It's mainly a philosophical question at this point, and those are often the most difficult ones in software design. :/

4.1 Serialize operator conventions

- If the type is polymorphic, it **must** call dest.impl_class(), passing it the string form of *this* type's class name. This is currently the only 100% reliable way to get the proper class names of your Serializable subtypes for use during during deserialization. (This is made clearer later via examples.) Monomorphic types can be reliably given a name by the framework, and normally no impl_class() needs to be called for them (SAM does this - section 13).
- Should save the object's state to the destination node, presumably using dest's API and the s11n functors/algorithms designed for such operations. State-saving may continue recursively for Serializable child objects.
- Returns true on success, false on error.

4.2 Deserialize operator conventions

- Should restore the state of an object via the node it is given, plus any sub-nodes, if needed. State-restoration may continue recursively for collecting Serializable child objects.
- The core library s11n generally makes sure that nodes are passed to objects of the types which serialized the nodes, but users may "brute-force" any node into any Serializable if they wish to. It is not the job of the deserialize operator to check that it has received a node for the proper type. It may do so, if it wishes, but this is out of line with s11n conventions, and not recommended.
- The core library only calls the deserialize operator *one time per object*, but it is possible that client code will trigger it multiple times for a given object. Thus any lists, pointers and whatnot should be cleaned up before restoring an object's state, to avoid problems like leaking resources or duplicating container entries.
- Returns true on success, false on error.

4.3 Data Node class names (impl_class()) (IMPORTANT!)

The importance of this method cannot be understated.

Let us repeat that many times:

```
while( ! this->gets_the_point() )
    std::cout << "The importance of impl_class() in the s11n framework cannot be understated.\n";
```

(Don't be ashamed if your loop runs a little longer than average. It's a learning process.)

impl_class() is part of the Data Node interface, and is used for getting and setting the class name of the *type* of object a node's data represents. This class name is stored in the meta-data of a node and is used for classloading the proper implementation types during deserialization. By convention the impl_class() is the string version of the C++ class name, including any namespace part, e.g., "foo::bar::MyClass". The library does not enforce this convention, and there are indeed cases where using aliases can simplify things or make them more flexible. See the class_loader documentation for hints on what aliasing can potentially do for you (see also lib/cl/src/cl_demp.cpp).

Client code *must*, unfortunately, call impl_class(), but the rules are very simple:

- Serializables (or their proxies) must call the target node's impl_class() in their *serialize operator* (not the deserialize operator), passing it the string name which the client code will later expect to be able to load the class with.
- If a Serializable class inherits serializable behaviour from a parent type, the subclass must set impl_class() *after* calling the parent implementation, to ensure the proper subclass type gets into the node.
- As of version 0.8.0, s11n can reliably set the class names for *monomorpic* and *base-most* types, but *cannot* do so for *polymorphic* types. In practical terms, this means that when proxying a monomorphic type, impl_class() does not need to be set by the client. For polymorphic types it must be set from within that type's serialize operator, as described above.

Algorithms which which parse data directly from data nodes irrespective of the node's impl_class() may ignore the impl_class(). One example is the de/serialize_streamable_xxx() family of functions: they use "raw" data nodes, to avoid a number of problems involved with registering proper class *names* for arbitrary containers' classloaders.

For more on class names, including how to set them in a uniform way for arbitrary types, see section 12.

4.3.1 Example impl_class() usage

Here's a sample which shows you all you need to know about the bastard child of the s11n framework, impl_class():

Assume class A is a Serializable base type using the *Default Serializable Interface* and B is a subtype of A. In A's *serialize* (*not* DEserialize) operator we must write:

```
node.impl_class( "A" );
```

In B's we should do:

```
if( ! this->A::operator()( node )[16] ) return false;
node.impl_class( "B" );
```

It is not strictly necessary that a subtype return false if the parent type fails to serialize, but it is a good idea unless the subtype knows how to detect and recover from the problem.

Follow those simple rules and all will be well when it comes to loading the proper type at deserialization time. To extend the above example. After the node contains B's state, we can do this:

```
A * a = s11nlite::deserialize<A>( node ); // we use A because that's the Base Type we're referencing on.
```

That creates a (B*), and deserializes it using B's interface.

Let's quickly look at two similar variants on the above which are generally not correct:

```
B * a = s11nlite::deserialize<A>( node );
```

That won't work - it will fail to compile because there is no implicit conversion possible from A to B. That one is straightforward, but the details for this one are fairly intricate:

```
B * a = s11nlite::deserialize<B>( node );
```

This will not fail to compile, but will probably not do what was expected. In this example B is now the "BaseType" for classloading/deserialization, which has subtle-yet-significant side-effects. For example, if B is never registered with *the B classloader* (e.g., class_loader<B>) then the user will probably be surprised when the above returns 0 instead of a new, freshly-deserialized object. If B is indeed registered with B's classloader, and B (as a standalone type) is recognized as a Serializable, then that call would work as expected: it would return a deserialized (B*).

4.3.2 Using local library support for impl_class()

Some heavily object-oriented libraries, like Qt (www.trolltech.com), support a polymorphism-safe className() function, or similar, where base types can get the proper class name of a subtype. If your trees support this, *take advantage of it*: set the impl_class() one time in the base type if you can get away with it! The sad news is, however, that the vast majority of us mortals must get by with doing this one part the hard way. :/ There are actually interesting macro/template-based ways to catch this for "many" use-cases, but no known 100% reliable way to catch them all.

4.4 Cooperating with other Serializable interfaces

Despite common coding practice, and perhaps even common sense, client Serializables *should not* (for reasons of form and code reusability) call their own interfaces' de/serialize methods directly! Instead they should use the various de/serialize functions. This is to ensure that interface translation can be done in s11n, allowing Serializables of different ancestries and interfaces to transparently interoperate. It also helps keep your code more portable to being used in other projects which support s11n. There are *exactly two* known cases where a client Serializable must call it's direct ancestor's de/serialize methods directly, as opposed to through a proxy: as the first operation in their de/serialize implementations. In those two cases it's perfectly acceptable to do so, and in fact could not be done any other way. Any other direct calls to a Serializable interface can be considered "poor form" and "unportable." If you find yourself directly calling a Serializable's de/serialize methods, see if you can do it via the core API instead (tip: *you can!*).

For example, instead of using this:

```
myserializable->serialize( somenode ); // NO! Poor form!  Unportable!
```

[16] See section 4.4 for why you should never directly call a Serializable's API. This particular case is one of two which simply cannot be avoided.

use one of these:

```
s11nlite::serialize( my_data_node, *myserializable ); // YES! Portable
s11n::serialize( my_data_node, *myserializable ); // Fine!
```

Note that there are extremely subtle differences in the calling of the previous two functions: the exact template arguments they take are different. In this case C++'s normal automatic argument-to-template type resolution suffices to select the proper types, so specifying them in <> is unnecessary. One *theoretical* exception is if the Data Node type is polymorphic, in which case ... email the dev list and we'll start a discussion about the potential implications ;).

In terms of Style Points (section 3.1), calling a Serializable's API directly, except in one of the two "that's-allowed" cases is immediately worth a good -2 SP or more, and may forever blemish one's reputation as a generic coder. Remember: except for the two exceptions mentioned above there is never anything you can do with the local API which you cannot also do with the "shared" API (barring, of course, the case of an expanded local s11n interface).

4.5 Member template functions as serialization operators

If a Serializable type implements template-based serialization operators, e.g.:

```
template <typename NodeType> bool operator()( NodeType & dest ) const;
template <typename NodeType> bool operator()( const NodeType & src );
```

then their de/serialize operators will support any NodeType supported by s11n. Note that s11nlite hides the abstractness of the NodeType, so users wishing to do this will have to work more with the core functions (which essentially only means using NodeType a *lot* more, e.g., `functioname<NodeType...>()`).

Using member template functions has other implications, and should be well-thought-out before it is implemented:

- May require puting the implementation in the header file, to avoid potential linking problems.
- Compilers do not completely check template functions until they are called, so there might be a compile-error-in-waiting for specific NodeTypes.
- Member function templates cannot be virtual. (This is a C++ restriction, not s11n-imposed.)

Despite those seeming limitations, experience suggests more and more that templated de/serialize operators generally offers more flexibility than non-templated. In the case of monomorphic types and proxies, there is almost never a reason to *not* make these operators member templates, and there are several *good* reasons to do so:

- The class can work with any Data Node type, instead of just, e.g., s11nlite::node_type.
- This is the only effective way to proxy requests for class templates, e.g., STL containers, as it allows a single pair of operators to handle de/serialization for a whole family of types. e.g., list<int>, list<double>, list<char> ... Note that this has nothing directly to do with the NodeType itself, but more with getting the impl_class() of the Serializable type.

5 How to turn JoeAverageClass into a Serializable...

In short, creating a Serializable is normally made up of these simple steps:

1. Create the class, implementing a pair of de/serialize methods with the signatures expected by s11n.
2. Tell s11n that your class exists, via registering it - see section 9.

If you are proxying a well-understood data structure for which a functor already exists to de/serialize it, step one complete disappears! An example would be proxying a std::list<int> or std::list<Serializable*> - those are both handleable by the s11n::list::list_serializer_proxy class, provided that the contained types are Serializables. For some useful proxy functors see section 10.

5.1 Create a Serializable class

The interface is made up two de/serialize operators. For this example we will use the so-called *Default Serializable Interface*, made up of two overloaded operator()s.

Assume we've created these classes:

```
class MyType {

    public:
    virtual bool operator()( s11nlite::node_type & dest ) const; // serialize
    virtual bool operator()( const s11nlite::node_type & src ); // deserialize
    // ... our functions, etc.

};
class MySubType : public MyType {

    public:
    virtual bool operator()( s11nlite::node_type & dest ) const; // serialize
    virtual bool operator()( const s11nlite::node_type & src ); // deserialize
    // ... our functions, etc.

};
```

It is perfectly okay to make those operators member function *templates*, templatized on the NodeType, but keep in mind that member function templates may not be virtual. Implementing them as templates will make the serialization operators capable of accepting any Data Node type supported by s11n, which may have future maintenance benefits.

If a Serializable will not be proxied, as the ones shown above are not, we must register it as being a Serializable, as shown here:

The base-most type is registered like so:

```
#define S11N_TYPE MyType
#define S11N_NAME "MyType"
#include <s11n/reg_serializable.h>
```

The subtype is registered like so:

```
#define S11N_TYPE MySubType
#define S11N_BASE_TYPE MyType
#define S11N_NAME "MySubType"
#include <s11n/reg_serializable.h>
```

For more information on the registration process, see section 9

If MyType does not support the default interface, see section 9.5 for instructions on registering it's interface with s11n.

5.2 Specifying Serializer Proxy functors

This is one of s11n's most powerful features. With this, *any* type can be made serializable, provided it's API is such that the desired data can be fetched and later restored. Almost all modern objects (those worth serializing) are designed this way, so this is practically never an issue.

Continuing the example from the previous section, if MyType cannot be made Serializable - if you can't, or don't want to, edit the code - then s11n can use a functor to handle de/serialize calls.

First we create a proxy, which is simply a struct or class with this interface:

Serialize:

```
bool operator()( data_node & dest, const SerializableType & src ) const;
```

Deserialize:

```
bool operator()( const data_node & src, SerializableType & dest ) const;
```

Notes about the operators:

- Yes, both functions "should probably" be const in this case, for the widest functor reusabiliity, but if C++ will let you get away with non-const operators in your contexts then s11n will accept them.
- They may be templated, and/or the functor may be templated. As long as C++'s normal type resolution can figure out what to do, it's legal.
- There are rare cases where calls can be ambiguously for this interface, so two functors - one each for de/serialization - may be necessary. In practice this is rare, however.

We must then register the proxy, as explained in section 9.6.

It may be interesting to know...

- There can be only one de/serialization handler for any given type, so you may not register both a Base Type and a proxy as being the handler for that Base Type, nor may you register two proxies as being the proxy for a single base type. Internally chaining calls within proxies can be used to get around the one-proxy limitation.
- Proxies may not normally save/load private data of the being-proxied type. In practice is is rarely an issue, as most modern libraries provide adequate accessors for their data. Classes designed such that they only possible way to store/restore their state is from internally should probably be redesigned to be more friendly. As a base-line comparison: every STL data structure which has been tried with this library has the necessary API to support proxying.
- A proxy class does not need to register with a classloader. It may be registered - there is no harm in doing so, but there is never a need to[17]. BaseType, on the other hand, must *always* be registered with the classloader.
- Proxies have a fixed interface - the function names and signatures may not be changed or marshaled (as Serializable interfaces can), for the simple reason that the proxies *are the ones doing the marshaling*.
- It may sometimes be necessary, due to ambiguity, to split a de/serialization functor into two functors.
- Proxies can potentially chain calls to each other together, which allows some interesting possibilities and very flexible control over de/serialization without touching your classes. e.g., a data versioning system could be implemented as a proxy which verifies a version property and then passes on the call to either the local Serializable interface of the object or to another proxy.
- Client code can, e.g., use a macro to define which proxy will be used for a given type, allowing them to switch freely between serialization implementations on a per-type basis.

i have a feeling there are a wide range of as-yes-undiscovered tricks for serialization proxies. Gary Boone calls this feature "s11n's most powerful," and i can't help but agree with him.

6 How to turn JoeNonAverageClass into a Serializable...

The techniques covered in the previous section work for most classes, but are not suitable for some others.

The following process works the same way for all types, as long as:

- The type implements a serializable interface we can register with s11n.

or:

- A functor can be registered which will take over serialization for the type.

All that must be done is that the desired interface must be registered with s11n, as described in section 9.

Note that when registering template types, you also need to register their *contained* types - they will be passed around just like other Serializables, so if s11n doesn't know about them you will get compile errors. And keed in mind that, e.g., list<int> and list<int*> *are different types*, and thus require different specializations. However, list<int> and (list<int>*) are equivalent for most of s11n's purposes.

Note that as of 0.8.x most standard containers need no special registration, with the exception that their contained types must be recognized Serializables, as mentioned above. Thus, list<vector<map<int,string> > > requires no registration whatsoever, but list<MySerializable> requires that MySerializable be a recognized Serializable type. Once that is done, a container type such as vector<list<map<string,MySerializable> > > can be transparently handled by the core.

[17] Or, more correctly, if you understand the *highly unusual* (and theoretical) case that would warrant such registration, then you'll understand why i oversimplify here ;).

7 Doing things with Serializables

Once you've got the Serializable "paperwork" out of the way, you're ready to implement the guts of your serialization operators. In s11n this is normally *extremely* simple. Some of the many possibilities are shown below.

In maintenance terms, the serialization operators are normally the only part of a Serializable which must be touched as a class changes. The "paperwork" parts do not change unless things like the class name or it's parentage change.

7.1 Setting "simple" properties

Any data which can be represented as a string key/value pair can be stored in a data node as a property:

```
node.set( "my_property", my_value );
```

set() is a function template and accepts a string as a key and any *Streamable Type* as a value.

There are rare cases involving ambiguity between ints/bools/chars which may require that the client explcitely specify the property's type as a template parameter:

```
node.set<int>( "my_number", mynum );
node.set<bool>( "my_number", mybool );
```

Use get_bool() if you wish to treat strings like "true", "1" and "yes" as equivalent to boolean true.

See the s11n::data_node API for the full getter/setter API.

Each property within a node has a unique key: setting a property will overwrite any other property which has the same key.

7.2 Getting property values

Getting properties from nodes is also very simple. In the abstract, it looks like:

```
T val = node.get<T>( "property_name", some_T_object );
```

e.g.,

```
this->name( node.get( "name", this->name() ) );
```

What this is saying is:

> Set this object's name to the value of the 'name' property of node. If 'name' is not set in the node, or cannot be converted to a string via i/o streams, then use the current value of this->name().

That sounds like like a mouthful, but it's very simple: when calling get() you must specify a second parameter, which must be of the same type as the return result. This second parameter serves several purposes:

- A default value: a known-good value to use in case the requested property is not set or could not be converted.
- An error value: The library cannot know what is an is not a valid value for such conversions, so the client may supply one here and compare it to what they expect. e.g., data versioning checks could be implemented this way.
- It tells get() what type of object it returns, without you having to specify get<ReturnType>("mykey").

As with set(), get() is a family of overloaded/templated functions, and there are cases where, e.g., int and bools may cause ambiguity at compile time. See set() for one workaround.

7.2.1 Simple property error checking

Here's how one might implement simple error checking for properties:

```
int foo = node.get( "meaning_of_life", -1 );
if( -1 == foo ) { ... error ... }
string bar = node.get( "name", "" );
if( bar.empty() ) { ... error ... }
```

Keep in mind that s11n cannot know what values are acceptable for a given property, thus it can make no assumptions about what values might be error values. In the case that there is literally no known error value for a property, but we *must* know whether it is set, we can either use node.is_set() or - more trickily - read the property twice using two different default values. If get() returns two different values on two successive calls then the property either is not set or is failing to convert via it's istream>> operator.

7.2.2 Saving custom Streamable Types

This is a no-brainer. Streamable Types are supported using the same get/set interface as all other "simple" properties. e.g., to save it:

```
node.set( "geom", this->geometry() );
```

and to load it:

```
this->geometry( node.get( "geom", this->geometry() ) );
```

or maybe:

```
this->geometry( node.get( "geom", Geometry() ) );
```

7.3 Finding or adding child nodes to a node

Use s11n::find_child_by_name() and s11n::find_children_by_name() to search for child nodes within a given node. Alternately, use the node's children() function to get the list of it's children, and search for them using a criteria of your choice. Keep in mind that in a deserialize operator, the node object will be const, and you must therefor declare the return value of these functions to be (const NodeType *). Failing to do so may cause an odd compile error.

Use s11n::create_child() to create a child and add it to a parent in one step. Alternately, add children using node.children().push_back().

7.4 Serializing Value Containers

Value Containers are, in this context, std::list- and std::map-compliant containers for which all stored types are Streamable Types (see 3.1). s11n can save, load and convert such types with unprecedented ease.

Normally containers are stored as sub-nodes of a Serializable's data node, thus saving them looks like:

```
s11n::map::serialize_streamable_map( targetnode, "subnode_name", my_map );
```

To use this function directly on a target node, without an intervening subnode, use the two-argument version without the subnode name. Be warned that none of the serialize_xxx() functions are meant to be called repeatedly or collectively on the same data node container. That is, each one expects to have a "private" node in which to save it's data, just as a full-fledged Serializable object's node would. Violating this may result in mangled content in your data nodes.

Loading a map requires exactly two more characters of work:

```
s11n::map::deserialize_streamable_map( targetnode, "subnode_name", my_map );
```

(Can you guess which two characters changed? ;)

If you want to de/serialize a std::list or std::vector, use the de/serialize_streamable_list() variants instead:

```
s11n::list::serialize_streamable_list( targetnode, "subnodename", my_list );
```

Note that s11n does not store the exact type information for data serialized this way, which makes it possible to convert, e.g., a std::list<int> into a std::vector<double *>, via serialization. The wider implication is that any list- or map-like types can be served by these simple functions (all of them are implemented in 6-8 lines of code, not counting typedefs).

7.4.1 Trick: "casting" containers

If you have lists or maps which are similar, but not exactly of the same types, s11n can act as a middleman to convert them for you. Assume we have the following maps:

```
map<int,int> imap;
map<double,double> dmap;
```

We can convert imap to dmap like this:

```
s11n::s11n_cast( imap, dmap );
```

Doing the opposite conversion "should" also work, but would be a potentially bad idea because any post-decimal data of the doubles would be lost upon conversion to int. Your compiler may or may not complain about that and may bail out, depending on the error/warning levels you have told your compiler to use.

7.5 De/serializing Serializable objects

In terms of the client interface, saving and restoring Serializable objects is slightly more complex than working with basic types (like PODs), primarily because we must deal with more type information.

7.5.1 Individual Serializable objects

The following C++ code will save any given Serializable object to a file:

```
s11nlite::save( myobject, "somefile.whatever" );
```

this will save it into a target data_node:

```
s11nlite::serialize( mynode, myobject );
```

The node could then be saved via an overloaded form of save().

There are several ways to save a file, depending on what Serializer you want to use. s11nlite uses only one Serializer by default, so we'll skip that subject for now (tip: see data_node_serialize.h and *_serializer.h for more detail, and s11nlite::serializer_class() for a way to override which Serializer it uses).

To load an object is fairly straightforward. The simplest way is:

```
BaseType * obj = s11nlite::load_serializable<BaseType>( "somefile.s11n" );
```

BaseType must be a type registered with the appropriate classloader (i.e., the BaseType classloader) and must of course be a Serializable type. To illustrate that first point more clearly:

```
SubTypeOfBaseType * obj = s11nlite::load_serializable<BaseType>( "somefile.s11n" );
```

It is critical that you use the base-most type which was registered with s11n, or you will almost certainly not get back an object from this function.

If you have a non-pointer type which must be populated from a file, it can be deserialized by getting an intermediary data node, by using something like the following:

```
s11nlite::node_type * n = s11nlite::load_node( "somefile.s11n" );
```

or:

```
const s11nlite::node_type * n = s11n::find_child_by_name( parent_node, "subnode_name" );
```

Then, assuming you got a node:

```
bool worked = s11nlite::deserialize( *n, myobject );
delete( n ); // NOT if you got it from another node! It belongs to the parent node!
```

Note, however, that if the deserialize failed, that myobject might be in an undefined or unusable state. In practice this is *extremely rare*, but it may happen, and client code may need to be able to deal with this possibility.

7.5.2 Lists of Serializable pointers

Saving lists of Serializables can be done several ways. The simplest way is:

```
s11n::list::serialize_list( targetnode, srclist );
```

srclist can be any list/vector-style container which contains SomeSerializableType, regardless of whether it is a pointer or reference type.

To deserialize a list of children is just as easy:

```
s11nlite::deserialize_list( srcnode, targetlist );
```

Note that templates figure out the type of Serializable based on the value_type of targetlist.

These functions support any type which is "basically compatible" with std::list, including std::vector.

7.5.3 Maps of pointers or value types.

For Serializable maps use the s11n::map::de/serialize_map() functions.

Those functions work with std::multimap as well as std::map.

If file size is a concern, the s11n::map::de/serialize_streamable_map() functions produce leaner output but only work with i/ostreamable types. These two are, however, untested with std::multimap (if it doesn't work for you, please report it to us as a bug!).

7.5.4 "Brute force" deserialization

Any data node can be de/serialized into any given Serializable, provided the Serializable supports a deserialize operator for that *node type*. The main implication of this is that clients may force-feed any given node into any object, regardless of the meta-data type of the data node (it's impl_class()) and the Serializable's type. This feature can be used and abused in a number of ways, and one of the most common uses is to deserialize non-pointer Serializables:

```
if( const data_node * ch = s11n::find_child_by_name( srcnode, "fred" ) )
    s11nlite::deserialize( *ch, myobject );
```

The notable down-side of doing this, however, is this: if the [de]serialize operation fails then myobject may be in an undefined state. With pointer children, solving this problem is fairly simple: delete it and create from a known-good set of data. With a non-pointer the handling of the case may get trickier. Again: a) this is very client-specific, and b) in practice it is very, very rare for a deserialization to fail.

8 Walk-throughs: imlementing Serializable classes

This section contains some example of implementing real-world-style Serializables. It is expected that this section will grow as exceptionally illustrative samples are developed or submitted to the project.

There are several complete, documented examples in the source tree, e.g., client/sample/src/demo_struct.cpp.

8.1 Sample #1: Read this before trying to code a Serializable!

Here we show the code necessary to save an imaginary client-side Serializable class, MyType.

The code presented here could be implemented either in a Serializable itself or a in a proxy, as appropriate. The code is the same, either way.

In this example we are not going to proxy any classes, but instead we will use various algorithms to store them. The end effect is identical, though the internals of each differ slightly.

8.1.1 The data

Let's assume we have a class, MyType, with this rather ugly mix of internal data we would like to save:

```
std::map<int,std::string> istrmap;
std::map<double,std::string> dstrmap;
std::list<std::string> slist;
std::list<MyType *> childs; // child objects
size_t m_id;
```

Looks bad, doesn't it? Don't worry - this is a *trivial* case for s11n.

8.1.2 The serialize operator

Saving member data normally requires one line of code per member, as shown here:

```
bool operator()( s11nlite::node_type & node ) const
{
```

```
    node.impl_class( "MyType" ); // critical!, but see below!
    node.set( "id", m_id );
    using namespace s11nlite;
    s11n::list::serialize_streamable_list( node, "string_list", slist );
    s11n::list::serialize_list( node, "child", childs );
    s11n::map::serialize_streamable_map( node, "int_to_str_map", istrmap );
    s11n::map::serialize_streamable_map( node, "dbl_to_str_map", dstrmap );
    return true;
}
```

A note about the "streamable" functions: we could use, e.g., serialize_list() instead of serialize_streamable_list(), but that form produces more verbose output.

As of 0.8.0, setting the impl_class() is not necessary for *monomorphic* types - for these types s11n can (accurately) collect the class name from the class registration information. For polymorphic types, however, this must be manually set (sorry!). See section 12 for more information.

8.1.3 The deserialize operator

The deserialize implementation is almost a mirror-image of the serialize implementation, plus a couple lines of client-dependent administrative code (not always necessary, as explained below):

```
bool operator()( const s11nlite::node_type & node )
{
    //////////////////// avoid duplicate entries in our lists:
    istrmap.clear();
    dstrmap.clear();
    slist.clear();
    s11n::free_list_entries( this->childs );
    //////////////////// now get our data:
    this->m_id = node.get( "id", m_id );
    s11n::list::deserialize_streamable_list( node, "string_list", slist );
    s11n::list::deserialize_list( node, "child", childs );
    s11n::map::deserialize_streamable_map( node, "int_to_str_map", istrmap );
    s11n::map::deserialize_streamable_map( node, "dbl_to_str_map", dstrmap );
    return true;
}
```

A note about cleaning up *before* deserialization:

In practice these checks are normally not necessary. libs11n never, in the normal line of duty, directly calls the deserialize operator more than *one time* for any given Serializable. It is conceivable, however, that client code will initiate a second (or subsequent) deserialize for a live object, in which case we need to avoid the possibility of appending to our current properties/children, and in the above example we avoid that problem by clearing out all children and lists/maps first. In practice such cases only happen in test/debug code, not in real client use-cases. The possibility of multiple-deserialization *is* there, and it is potentially ugly, so it is prudent to add the extra few lines of code necessary to make sure deserialization starts in a clean environment.

8.1.4 Serializable/proxy registration

The interface must now be registered with s11n, so that it knows how to intercept requests on that type's behalf: for full details see section 9, and for a quick example see 6.

8.1.5 Done! Your object is now a generic Serializable Type!

That's all there is to it. Now MyType will work with any s11n API which work with Serializables. For example:

```
s11nlite::save( myobject, std::cout );
```

will dump our MyObject to cout via s11n serialization. This will load it from a file:

```
MyType * obj = s11nlite::load_serializable<MyType>( "filename.s11n" ); // also has an istream overload
```

(Keep in mind that the object you get back might actually be some ancestor of MyType - this operation is polymorphic if MyType is.)

Now that wasn't so tough, was it?

A very significant property of MyType is this:

> MyType is now inherently serializable by *any code which uses s11nlite*, regardless of the code's local Serialization API! s11n takes care of the API translation between the various local APIs.

Weird, eh? Let's take a moment to day-dream:

Consider for a moment the *outrageous* possibility that 746 C++ developers worldwide implement s11n-compatible Serializable support for their objects. Aside from having a convenient serialization library at their disposal (i mean, *obviously* ;), those 746 developers now have *100% transparent* access to each others' serialization capabilities.

Now consider for a moment the implications of *your* classes being in that equasion...

Let us toke on that thought for a moment, absorbing the implications.

Well, ***i*** think it's pretty cool, anyway.

8.2 Gary's Code (a.k.a. "The Dream")

> **TIP:** this section has some very informative, revealing information about:

1. The classloader's role in s11n.
2. Tips, hints and tricks for developing your own Seriaizables and proxies.
3. Why s11n, as a problem domain, is so interesting to it's daddy. i.e., WTF i spend so much time working on this code. ;)

The code in this example, and much of the commentary accompanying it, was submitted by ***Gary Boone***. Gary is one of the first client-side users to show an interest in s11n. His feedback and interaction has been truly intstrumental in driving some of the recent changes (i.e., 0.8.x), particularly in regards to usage simplifications and improved documentation (from build docs to this manual - the whole gamut).

> ***THANKS, GARY!!!*** You're a *great* example of how user feedback can *directly* , and *notably*, affect the development of Open Source projects!

8.2.1 Background context and some longer-term history

Gary has been trying to save a container of structs, each containing a couple POD types. As anyone who has attempted such a thing at the stream level can tell you, even for relatively trivial containers and data types (e.g., even non-trivial strings):

> Saving data is relatively easy. Loading data, especially via a generic interface, is ***mind-numbingly, ass-kickingly difficult!***

The technical challenges involved in loading even relatively trivial data, *especially* trying to do so in a unified, generic manner, are *downright frigging scary*. Some people get their doctorates trying to solve this type of problem[18]. Complete *branches* of computer science, and hoardes of computer scientists, students, and acolytes alike, have researched these types of problems for practically *eons*. Indeed, their efforts have provided us a number of *critical* components to aid us on our way in finding the Holy Grail of serialization in C++...

IOStreams, the predecessor of the current STL iostreams architecture[19], brought us, the C/C++ development community, *tremendous* steps forward, compared to the days of reading data using classical brute-force techniques provided by standard C libraries. That model has evolved further and further, and is now an instrumental part of almost any C++ code[20], but the practice of directly manipulating data via streams is showing its age. Such an approach is, more

[18] But all i got was this library manual. ;)

[19] That was well before my time, but i read a lot of C++ books. ;)

[20] Are you going to tell me you never use std::cout and std::cerr? Yeah, right. Tell it to your grandma - maybe she'll believe you.

often than not, not suitable for use with the common higher-level abstractions developers have come to work with over the past decade[21].

In the mid-1990's HTML become a world-wide-wonder, and XML, a more general variant from same family of meta-languages HTML evolved from, SGML[22], leapt into the limelite. Pratically overnight, XML evolved into *the* generic platform for data exchange and, even more significantly, *conversion and interchange*. XML is here to stay, and i'm a *tremendous* fan of XML, but XML's era has left an even more important legacy than the elegance of XML itself:

More abstractly, and more fundamentally, the popularity and "well-understoodedness" of XML has *greatly* hightened our collective understanding of abstract data structures, e.g. DOMs [Document Object Models], and our understanding of the general needs of data serialization frameworks. *That latter point should be neither overlooked nor underestimated!*

What time is it now? 2004 already? It looks like we're ready for another 10-year cycle to begin...

We're in the 21st century now. In languages like Java(tm) and C# serialization operations are basically built-in (though i do have very deep fundamental differences with Java's whole serialization model!). Generic classloading, as well, is EASY in those languages. Far, far away from Javaland, the problem domain of loading and saving data has terrified C++ developers *for a full generation*!

s11n aims, rather ambitiously, to put an end to that. The whole general problem of serialization is a very interesting problem to me, on a personal level. It fascinates me, and s11n's design is a direct result of the energy i have put into trying to rid the world of this problem *for good*.

Well, okay, i didn't honestly do it to save the world['s data]:

i want to save my objects!

That's my dream...

Oh, my - what a coincidence, indeed...

That's s11n dream, too!

s11n is *actively* exploring viable in-language C++ routes to *find*, then *take*, the C++ community's *next major evolutionary step* in general-purpose object serialization... all right at home in ISO-standard C++. This project takes the learnings of XML, DOMs, streams, functors, class templates, Meyers, Alexandrescu, Strousup, Sutter, Dewhurst, "Gamma, et al", comp.lang.c++, application frameworks, PHP, Java[23], and... even lowly ol' me - yeah, i'm the poor bastard who's been pursuing this problem for 3+ years ;).

In short, s11n is attempting to apply the learning of an entire generation of software developers and architects, building upon of the streets they carved for us... through the silicon... armed only with their bare text editors and the source code for their C compilers. These guys have my *utmost* respect. Yeah, okay... even the ones who chose to use (or implement!) vi. ;)

Though s11n is quite young, it has a years-long "conceptual history"[24], yet it's capabilities *far* exceed any original plans i had for it. Truth be told, i use it in all my C++ code. i can finally... *finally*, ***FINALLY SAVE MY OBJECTS!!!!***

i hope you will now join me in screaming, at the loudest possible volume:

It's about damned time!!!

8.3 Meanwhile, back in the present day... (Gary's code, remember?)

Let us repeat the s11n mantra (well, one of several[25]):

s11n is here to ***Save Your Data****, man!*

[21] Things have changed ***a lot*** since the '80s. Consider for a moment if you could ever, in good conscience, go back to writing data containers and i/o code in C! No std::string, no std::map, no... std::, period!

[22] [Standard,Structured] Generic Markup Language

[23] Incidentally, not C#: s11n was started before i ever touched C#. In all honesty, i find C#'s core model to be inferior to s11n, at least in terms of it's client-side interface. For example, it really bugs me that in C# (or any other serialization framework), the client must know something so basic as what file format their data is stored in. i say (and s11n says): only a file's i/o parsers *really* care what format a file is in.

[24] Utility-class coding, and *lots* of design thought, started in early 2001. The "real coding" began in September, 2003, once i finally cracked the secrets i needed to implement my dream-classloader.

[25] Trivia note: The banner label on the s11n web site rotates through s11n's list of official mantra. New mantra are added as they ar discovered. Submit your s11n mantra or clever quip and it will show up on the s11n web site. :)

The type of problem Gary is trying to solve here is s11n's *bread and butter*, as his solution will show us in a few moments.

Now, back to Gary's story...

After getting over the initial learning hurdles - admittedly, s11n's abstractness can be a significant hinderness in understanding it - he got it running and sent me an email, which i've reproduced below with his permission.

i have made only minor changes to his example code, to fix a relatively minor ommission in his solution (but, all in all, *not bad* for someone just starting with s11n!). i must say, it gives me *great pleasure* to post Gary's text here. Through his mails i have witnessed the dawning of his excitement as he comes to understanding the general utility of s11n, and that is one of the greatest rewards i, as s11n's author, can possibly get. Reading his mail certainly made me feel good, anyway :).

Gary's email address has been removed from these pages at his request. If, after reading his examples, you're intested in contacting Gary, please send me a mail saying so and i will happily forward it on to him.

In the interest of explanation and example, the first part of Gary's text below is posted as he sent it - *with the ommision i mentioned a moment ago*. i will cover that ommision afterwards, by simpy pasting it in the way i explained it to Gary. In some places i have added descriptive or background information, marked like so:

> [editorial: ....]

8.3.1 *Gary's Revelation*

[From: Gary Boone, 12 March 2004]

...

> Attached is my solution ('map_of_structs.*'). Basically, I followed your suggestion of writing the vector elements as node children using a for_each & functor.
>
> ...
>
> I like the idea of not having to change **any** of my objects, but instead use functors to tell s11n how to serialize them.
>
> ...
>
> Dude, it works!! That's amazing! That's huge, allowing you to code serialization into your projects without even touching other people's code in distributed projects. It means you can experiment with the library without having to hack/unhack your primary codebase.
>
> Stephan, you **have** to make this clearer in the docs! It should be example #1:

[editorial: i feel compelled to increase the font size of that last part by a few points, because i had the distinct impression, while reading it, that Gary was overflowing with amazement at this realization, just as i first did when the implications of the archtecture started to trickle in. :) That said, the full implications and limits of the architecture not yet fully understood, and probably won't be in the forseeable future - i honestly believe it to be *that* flexible.]

> ...
>
> One of the most exciting aspects of s11n is that you may not have to change **any** of your objects to use it! For example, suppose you had a struct:
>
> struct elem_t {
>
> ```
> int index;
> double value;
> elem_t(void) : index(-1), value(0.0) {}
> elem_t(int i, double v) : index(i), value(v) {}
> ```
>
> };
>
> You can serialize it without touching it! Just add this proxy functor so s11n knows how to serialize and deserialize it:
>
> // define a functor for serialization/deserialization of elem_t structs
>
> struct elem_t_s11n { // note: no inheritence *requirements*, but polymorphism is permitted.
>
> ```
> /************************************
> // a so-called "serialization operator":
> // This operator stores src's state into the dest data container.
> ```

```
    // Note that the SOURCE Serializable is const, while the TARGET
    // data node object is not.
    ************************************/
    bool operator()( s11n::data_node &dest, const elem_t &src ) const[26] {
        dest.impl_class("elem_t");
        dest.set("i", src.index);
        dest.set("v", src.value);
        return true;
        [editorial: the original code was missing that return. i didn't catch it until editing this
        manual.]
    }
    /************************************
    // a "deserialization operator":
    // This operator restores dest's state from the src data container.
    // Note that the SOURCE node is const, while
    // the TARGET Serializable object is not.
    ************************************/
    bool operator()( const s11n::data_node &src, elem_t &dest ) const {
        dest.index = src.get("i", -1);
        dest.value = src.get("v", 0.0);
        return true;
        [editorial: ditto regarding the return value]
    }
};
```

[editorial: while the similar-signatured overloads of operator() may seem confusing or annoying at first, with only a little practice they will become second nature, and the symmetry this approach adds to the API improves it's overall ease-of-use. Note the bold text in their descriptions, above, form simple pneumonics to remember which operator does what.
The constness of the arguments ensures that they cannot normally (i.e., via standard s11n operations) be called ambiguously. That said, i have seen one case of a proxy *functor* (not Serializable) for which call-ambiguity was been a problem, which is why proxies *may* optionally be implemented in terms of two objects: one SerializeFunctor and a corresponding DeserializeFunctor, each of which must implement their corresponding halves of the de/serialize equasion. Often it is very useful to first implement de/serialize *algorithms* (i.e. *as functions*) and then later supply the 8-line wrapper *functor* class which forwards the calls to the algorithms. Several internal proxies do exactly this, and it gives client code two different ways of doing the same thing, at the cost of an extra couple minutes of coding the proxy wrapper around your algoritm. As a general rule, algorithms are slightly easier to test early on in the development, as they are missing one level of indirection which proxies logically bring along.
Back to you, Gary...]

The final step is to tell s11n about the association between the proxy and it's delegatee:

```
#define S11N_TYPE elem_t
#define S11N_TYPE_NAME "elem_t"
#define S11N_SERIALIZE_FUNCTOR elem_t_s11n
#include <s11n/reg_proxy.h>
```

[editorial: Gary's original code, for 0.7.x, was replaced here with the 0.8.x method, to avoid confusion. The effect is the same.
After this registration, `elem_t_s11n` is now *the* official delegate for *all* de/serialize operations involving `elem_t`, or for any recognized/registered sub-type of `elem_t`[27]. Any time a de/serialize operation involves an `elem_t` or (`elem_t *`) s11n will direct the call to `elem_t_s11n`. The *only* way for a client to bypass this proxying is to do the most *dispicable*, *unthinkable* act in all of libs11n: passing the node to the Serializable directly, using the Serializable's API! See section 4.4 for an explanation of why taking such an action is considered *Poor Form*.]

[26] Whether or not a functor has const or non-const operator()s is largely a matter of what the functor is used for. The constness of the *arguments* is *set* - they may not deviate from that shown here. The constness of the operator itself is not defined by s11n conventions.

[27] This sample class is monomorphic, but the exact same conventions apply for polymorphic Serializable types.

You're done. Now you can serialize it as easily as:

```
elem_t e(2, 34.5);
s11nlite::save(e, std::cout);
```

Deserializing from a file or stream is just as straightforward:

```
elem_t * e = s11nlite::load_serializable<elem_t>( "somefile.elem" );
```

or:

```
elem_t e;
bool worked = s11nlite::deserialize( node, e );
```

[editorial: that last example basically "cannot fail" unless elem_t's deserialize implementation *wants* it to, e.g., if it gets out-of-bounds/missing data and decides to complain by returning false. What might cause *missing* data in a node? That's exactly what would effectively happen if you "brute-force" a node populated from a non-elem_t source into an elem_t. Consider: the node will probably *not* be laid out the same internally (different property names, for example), and if it *is* laid out the same, there are still no guarantees such an operation is *symantically* valid for elem_t. Obviously, handling such cases is 100% client-specific, and must be analysed on a case-by-case basis. In practice this problem is purely theoretical/academic in nature: such a problem never happens. Consider: frameworks understand their own data models, and don't go passing around invalid data to each other. s11n's strict classloading scheme means it cannot inherently do such things, so that type of "use and abuse" necessarily comes from client-side code. Again: *this never happens.* Jesus, i'm so pedantic sometimes...]

...

[End Gary's mail]

Gary hit it right on the head. The above code is *exactly* in line with what s11n is designed to do, and his first go at a proxy was implemented exactly correctly. Kudos, Gary![28]

HOWEVER... as mentioned earlier, there is a slight ommision in this example, which we'll cover next. It's the type of potential problem which could easily lie in waiting for a long time without being discovered... *yes*... ***that*** type of problem!

8.3.2 A minor, but significant, addition...

> **ACHTUNG: The info in this section has been partly obsoleted by newer registration techniques: separate, explicite, client-side classloader registration is no longer required. However, the text is still informative, and gives some (still-applicable) insights into s11n not found anywhere else.**

Gary's submission was, from an s11n perspective, flawlessly implemented, except that one tiny detail slipped by. Admittedly, it's probably a detail which only one person on the planet currently truly understands in all of it's intricacies - s11n's author.

Ironically (as we'll see soon), for Gary's particular use-case, the ommision he made (yes, i'll finally tell you in a moment was it is!) would never cause a "real" problem - i.e., client code would *mostly* work as expected - for reasons explained in detail below. Thus, Gary's code didn't have a bug, per se, but an *ommision*, which could potentially have turned into a bug someday (and a hard-to-find one, at that).

Here is my response to Gary's submission, edited in the interest of clarity, example... and sobriety level ;)

```
>> Gary wrote:
> // ...then tell s11n about it
> // register the proxy
> S11NLITE_PROXY(elem_t, elem_t_s11n, elem_t_s11n);
```

That's all ***perfect***, except for one tiny (but important) detail:

```
// register elem_t with it's classloader:
S11NLITE_CL_BASE(elem_t);
```

Everything *will* actually work *without* this registration *until* you try:

```
elem_t * e = s11nlite::deserialize<elem_t>( node );
```

[28] Let's call that +1 SP ;).

Then the elem_t classloader won't be able to find a class named "elem_t", i.e., the node's impl_class(). We know the impl_class() is "elem_t" because... (have you guessed yet?) ... the Serializable Proxy set that value in it's serialize operator - *exactly* what it was supposed to do. In s11n-process terms, it is always the job of a Serializable/proxy to set it's the impl_class() into the target serialization node. Exceptions are allowed when, e.g., a specific functor and algorithm are designed to work with one another, such that perhaps the algorithm actually takes over the impl_name() responsibility.

The so-called "brute-force" form of deserialization will still work without the classloader registration:

```
elem_t e;
s11nlite::deserialize( node, e );
```

Why? The answer is deceptively simple: let's consider what happens when this call is made:

We ask s11n to give node to e [e's proxy] so that e can restore it's state from the node. Ah... we already have a node. There's the answer: this approach simply hands the node *directly* over to e [e's proxy], *bypassing the need for a classloader*. Consider: we handed the node directly to an existing serializable, and thus we don't need to create a Serializable object before we populate it (as would be the for a call to deserialize<BaseType>(node)). Ergo... no classloader operation is directly invoked there. That said, if elem_t implements recurive deserialization (i.e., if it contains child Serializables), then a classloader call may be invoked by one of the sub-deserializations.

IMO those options (brute-force vs. deserialize-to-new-object) give all the de/serialization flexibility a Serializable needs, in terms of API interface - they can bypass the CL altogether if they like (a custom CL can be installed for any given BaseType, too... see data_node_serialize.h for info).

A longer version of the truth...

Truth be told, CL reg is not *always* necessary:

- If type T's been CL registered "somewhere else" (i.e., other code files, or even *in other libs we link to*) then *we don't need it*. (The classloader is nearly magical in this regard.)
- If an object is only "brute-force" deserialized, CL registration is also *not needed*.

[Not true for 0.8+, but still informative:]

The reason CL reg cannot be done as a part of the s11n-[Serializable/Proxy] registration macros is that it is too easy to get ODR [One Definition Rule] violations (happens all the time, actually). Thus the small burden of CL reg must be placed on the clients, who must simply ensure that no single type is registered with the CL more than once *per compilation unit*. In practical terms, that is easy to enforce, and the anonymous namespaces which the reg code live in play a *BIG* role in avoiding ODR collisions across multiple compilation units. In linking terms, there are LOTS of options for linking CL registrations into an app, as covered a bit in this manual, and in more detail in the libclass_loader manual.

Given the pros and cons, s11n takes the more cautious (and ultimately much more flexible) route of requiring that *someone* register the appropriate types with the CL - s11n will not do this by itself except for a small number of common types (PODs/string).

9 s11n registration & "supermacros" (IMPORTANT)

As of version 0.8.0, s11n uses a new class registration process, providing a single interface for registering any types, and handling all classloader registration.

Historically, macros have been used to handle registration, but these have a *huge* number of limitations. We now have a new process which, while a tad more verbose, is far, far superior is many ways (the only down-side being it's verbosity). i like to call them...

9.1 "Supermacros"

s11n uses generic "supermacros" to register anything and everything. A supermacro is a header file which is written to work like a C++ macro, which essentially means that it is designed to be passed parameters and included, potentially repeatedly.

Use of a supermacro looks something like this:

```
#define MYARG1 "some string"
#define MYARG2 foo::AType
#include "my_supermacro.h"
```

By convention, and for client convenience, the supermacro is responsible for unsetting any arguments it expects after it is done with them, so client code may repeatedly call the macro without #undef'ing them.

Sample:

```
#define S11N_TYPE std::map<std::string,std::string>
#define S11N_NAME "std::map<std::string,std::string>"
#define S11N_SERIALIZE_FUNCTOR s11n::value_map_serializer_proxy
#include <s11n/reg_proxy.h>
```

While the now-outmoded registration macros are (barely) suitable for many non-templates-based cases, supermacros allow some - er... *TONS* - of features which the simpler macros simply cannot come close to providing. For example:

- A supermacro can handle almost any case, using a single - yet extendable - interface, and more complex variants can implement their own "supermacro" file.
- Supermacros can do arbitrary tasks, like classloader registration, freeing clients of this task.
- Arbitrary new supermacros can be introduced at any time without impacting existing code, which means, for example, client code can use a #define to swtich between interfaces, by including different registration supermacros.
- ODR violations can be more easily eliminated (in theory, completely), as each supermacro is free to implement it's internals however it wants. e.g., if it uses a custom classloader registration technique then it cannot collide with the default implementations provided via libclass_loader's macros.
- As they are implemented in "real header code", they are completely immune to the typical limitations of macros, and simply *much* easier to maintain.
- This approach does *all* necessary registration in one step, including classloader registration (which could not be reliably done via the macro approach, due to potential of ODR-violations).
- Supermacros can be arbitrarily large, wheres macros get very tedious to edit once they are longer than a few lines.
- They are *much, much* easier to debug when something doesn't compile: we even get proper file names and line numbers (*yes!!!!*).
- At least a handful of significant maintenance benifits come to mind.

The adoption of the supermacro mechanic into s11n 0.8 opened up a huge number of possibilities which were simply not practical to do before, and the implications are still not fully appreciated/understood.

9.2 General: Base Types

All of s11n's activity is "keyed" to a type's Base Type. This is used for a number of internal mechanisms, far too detailed to even properly *summarize* here. A BaseType represents the base-most type which a "registration tree" knows about. In client/API terms, this means that when using a heirarchy of types, the base-most Serializable type should be used for all templatized BaseType/SerializableType parameters.

(See, it's difficult to describe!)

In most usage using BaseTypes as key is quite natural and normal, but one known case exists where they can be easily confused:

Assume we have this heirachy:

AType <–[extended by] – BType <– CType

In terms of matching BaseType to subtypes, for *most purposes, that looks like this:*

- BType's BaseType is AType
- CType's BaseType is ***AType***

There are valid cases where registering both bases of CType are useful, but doing so in the same compilation unit will fail with the default registration process, with ODR collisions. The need to do this is rare, in any case, and requires a good understanding of how the classloader works. Doing it is very straightforward, but requires a bit of client-side effort.

9.3 Choosing class names when registering

s11n does not care what class names you use. We could use the name "fred" for, e.g., std::map<string,string> and the end effect is the same as if we had used it's "real" name. In fact, we could also call the pair type contained in that map "fred" *without getting a collision* because those two types use different classloaders.

The important thing is that you are consistent with class names. Once you change them, any older data will not be loadable via the classloader unless you explicitely alias the type names: see cllite::alias() for how to do this, or see the example in reg_serializer.h.

By convention, s11n uses a class' C++ name, minus any const and pointer parts, as those parts are *irrelevant* for purposes of classloading and cause *completely unnecessary* maintenance in other parts of the code (including, potentially, client code). Thus, when s11n saves a (std::string *) and a (std::string), the type s11n uses will be "std::string" for *both* of them, and the context of a deserialization determines exactly which form is created.

9.4 Registering Base Types supporting serialization operator()s

As of s11n 0.8, s11n "requires" so-called Default Serializables to be registered. In truth, they don't *have* to be for all cases, but for widest compatibility and ease of use, it is highly recommended. It is pretty painless, and must be done only one time per type:

```
#define S11N_TYPE ASerType
#define S11N_NAME "ASerType"
#include <s11n/reg_serializable.h>
```

For a registration of a subtype of ASerType, use:

```
#define S11N_BASE_TYPE ASerType
#define S11N_TYPE BSerType
#define S11N_NAME "BSerType"
#include <s11n/reg_serializable.h>
```

The S11N_xxx macros are reset when including the registration code, so client code need not unset them before redefining them.

9.5 Registering types which implement a custom Serializable interface

If a class implements a pair of de/serialization functions, but does not use operator() overloads, the process is simply a minor extension of the default case described in the previous section.

For example, assume we have the following two member functions in our classes:

[virtual] bool save()(data_node & dest) const;

[virtual] bool load()(const data_node & src);

(The same names may be used for both functions, as long as the constness is such that they can be properly told apart by the compiler.)

Simply add these two defines before including the registration supermacro:

```
#define S11N_SERIALIZE_FUNCTION save
#define S11N_DESERIALIZE_FUNCTION load
```

That's it - you're done.

Note that it is okay to pass `operator()` as the function names, but doing so is redundant - this is the default behaviour.

9.6 Registering Serializable Proxies

In fact, there is no one single way to do this, because there are several pieces to a registration:

The important things are:

- Proxied type must be registered with appropriate classloader (normally it's own).
- Proxied class' name should be registered with class_name<ProxiedType>. Not strictly required, but very useful.

- Proxy implementation must be have a SAM specialization installed (section 13).

After months of experimentation, s11n refines the process to simply calling the following supermacro:

```
#define S11N_TYPE ASerType
#define S11N_NAME "ASerType"
#define S11N_SERIALIZE_FUNCTOR SampleProxySerializer
// optional:  #define S11N_DESERIALIZE_FUNCTOR SampleProxyDeserializer
// DESERIALIZE defaults to the SERIALIZE functor, which works fine for most cases.
#include <s11n/reg_proxy.h>
```

This is repeated for each proxy/type combination you wish to register. The macros used by reg_proxy.h are temporary, and unset when it is included.

There are other optional macros to define for that header: see reg_proxy.h for full details.

If we extend ASerType with BSerType, B's will look like this:

```
#define S11N_BASE_TYPE ASerType
#define S11N_TYPE BSerType
#define S11N_NAME "BSerType"
#include <s11n/reg_proxy.h>
```

Without the need to specify the functor name - it is inherited from the BASE_TYPE.

9.7 Where to invoke registration (IMPORTANT)

It is important to understand exactly where the Serializable registration macros need to be, so that you can place them in your code at a point where s11n can find them when needed. In general this is very straightforward, but it is easy to miss it.

At any point where a de/serialize operation is requested for type T via the s11n core framework (including s11nlite), the following conditions must be met:

- The Serializable registration implementation code for T must be available to s11n. In practice, this means that the registration code must be available to the the client code requesting the operation at the time it is compiled.
- T must be a complete type, not, e.g., defined only via a forward declaration. (T's *implementation* need not be available, but it's *interface* must be declared.)

Because of s11n's templated nature, these rules apply *at compile time*. This essentially means that the registration should generally be done in one of the following places:

- T's header file. (Most straightforward, but also arguably the sloppiest and most intrusive on T.)
- The implementation file(s) making the operation. (Be careful to avoid unduly duplicating registrations, for maintenance reasons.)
- A separate header created exclusively for this purpose, which is included by any code which initiates de/serialize operations on T objects. For example, we might have T.h and T_s11n.h, which registers the class with s11n. (This is probably the cleanest solution for non-trivial projects.)

In the simplest client-side case - a main.cpp with all implementation code in that file - simply call the registration macros right after each class' declaration.

9.7.1 Hand-implementing the macro code (IMPORTANT)

Whenever these docs refer to calling a certain macro, what they *really* imply is: include code which is functionally equivalent to that generated by the macro. This code can be hand-written, generated via a script, or whatever. In any case, it must be available when s11n needs it, as described above.

10 Existing proxies, functors and algorithms

s11n's ability to use algorithms, functors and proxies to de/serialize arbitrary types is the heart of it's power and flexibility. The library comes with a number of useful functors/algos/proxies, some of which are described in this section. Once a couple of these are understood, implementing customized ones is very straightforward.

Most of the classes/functions listed below live in one of the following files:

lib/node/src/data_node_functor.h

lib/node/src/data_node_ago.h

lib/standalone/src/algo.h

lib/standalone/src/functor.h

The whole library, with the unfortunately exception of the Serializer lexers, is based upon the STL, so experienced STL coders should have no trouble coming up with their own utility functors and algorithms for use with s11n. (Please submit them back to this project for inclusion in the mainstream releases!)

10.1 Commonly-used Proxies

This section briefly lists some of the available proxies which are often useful for common tasks.

To install any of these proxies for one your types, simply do this:

```
#define S11N_TYPE MyType
#define S11N_NAME "MyType"
#define S11N_SERIALIZE_FUNCTOR serializer_proxy
// #define S11N_DESERIALIZE_FUNCTOR deserializer_proxy
// ^^^^ not required unless noted by the proxy's docs.
#include <s11n/reg_proxy.h>
```

In theory, passing an algorithm *function* name as the functor(s) will also work, but it hasn't been tested yet.

When writing proxies, remember that it is perfectly okay for proxies to hand work off to each other - they may be chained to use several "small" serializers to deal with more complex types. As an example, the pair_serializer_proxy can be used to serialize each element of any map. If you write any proxies or algorithms which are compatible with this framework, *please submit them to us!*

Keep in mind that most std:: containers are automatically proxied by the "most generic" proxy available. As usual, "most generic" also means "not the most efficient" for all cases. Clients may set up their own proxies for specific container instantiations. As of 0.8.3, the following containers are handled without any client-side intervention:

- std::list
- std::vector
- std::map and std::multimap
- std::pair
- std::set (std::multiset "might" work - it's untested)

10.1.1 Streamable types: s11n::streamable_type_serializer_proxy

This proxy can handle any streamable type, treating it as a single Serializable object. Thus a proxies int or float will be stored in it's own data node during serialization. While this is definately not space-efficient for small types, it allows some very flexible algorithms to be written based off of this functor, because PODs registered with this proxy can be treated as full-fledged Serialiables.

s11n installs this proxy for all basic POD types and std::string by default. Clients may plug in any i/ostreamable types they wish using the reg_proxy.h supermacro.

10.1.2 list/vector/set: s11n::list::list_serializer_proxy

This flexible proxy can handle any type of list/vector/set containing Serializables. It handles, e.g., list<int> and list<int*>, or vector<pair<string,MySerializable*> >, and set<string>, provided the internally-contained parts are Serializable. Remember, the basic PODs are inherently handled, so there is no need to register the contained-in-list type for those or std::string.

Trivia:

> The source code for this proxy shows an interesting example of how pointer and non-pointer types can be treated identically in template code, including allocation and deallocation objects in a way which is agnostic of this detail. This makes some formerly impossible-or-difficult cases very staightforward to implement in one function.

10.1.3 pair: s11n::map::pair_serializer_proxy

Like list_serializer_proxy, this type can handle pairs containing any pointer or reference type which is itself a Serializable. It would be highly unusual to use this proxy directly - it exists primarily to simplify the implementation of the std::map proxy.

10.1.4 map/multimap: s11n::map::map_serializer_proxy

Like list_serializer_proxy, this type can handle maps containing any pointer or reference type which is itself a Serializable. This proxy also works for std::set and std::multimap.

Alternately, maps containing only Streamable Types may be proxied by s11n::map::streamable_map_serializer_proxy. This proxy will produce leaner output, but is only suitable for Streamables and is untested with multimaps (if that doesn't work, it's a bug).

10.2 Commonly-used algorithms, functors and helpers

The list below summarizes some algorithms which often come in handy in client code or when developing s11n proxies and algorithms. Please see their API docs for their full details, and please do *not* use one of these without understanding it's conventions and restrictions.

More functors and algos are being developed all the time, as needed, so see the API docs for new ones which might not be in this list.

function() or functor	Short description
free_[list,map]_entries()	Generically deallocates entries in a list/map and empties it.
create_child()	Creates a named data node and inserts it into a given parent.
child_pointer_deep_copier	Deep-copies a list of pointers. Not polymorphism-safe.
object_deleter	Use with std::for_each(), to generically deallocate objects.
pointer_cleaner	Essentially a poor-man's multi-pointer auto_ptr.
de/serialize_streamable_map()	Do just that. Supports any map/multimap containing only i/ostreamable types.
de/serialize_streamable_list()	Ditto, for list/vector types.
de/serialize_[map/list/pair]()	de/serialize containers of Serializables.
object_reference_wrapper	Allows referring to an object as if it is a reference, regardless of it's pointerness.
pair_entry_deallocator	Generically deallocates elements in a pair<X[*],Y[*]>.
abstract_creator	A weird type to allow consolidation of some algos regardless of argument pointerness.

11 Data Formats (Serializers)

s11n uses an interface, generically known as the Serializer interface, which defines how client code initializes a load or save request, but specifies nothing about data formats. Indeed, the i/o layer of s11n is implemented on top of the core serialization API, which was written before the i/o layer was, and the core is 100% code-independent of this layer. In s11nlite only one Serializer is used by default: use s11nlite::serializer_class() to change it.

11.1 General conventions

However data-format agnostic s11n may be, all supported data formats have a similar logical construction. The basic conventions for data formats compatible with the s11n model are:

- Each data file contains, at most, one root node, per long-standing XML conventions.
- Nodes may represent any Serializable type, with all that that implies, or "raw" data nodes (i.e., nodes without meta-information specifying the type of object they represent).
- Nodes may contain an arbitrary number of child nodes.
- Nodes must have a name meeting the criteria specified in section 4.3. The name need not be unique.
- Nodes must have an "impl class" - the class name of the type for which the node contains data, to be used by the classloader when deserializing the node. It is acceptable to use "dummy names" here, provided someone knows how to parse the data out (e.g., the functions described in in section 7.4 work this way). s11n defines an impl_class() accessor function for getting and setting this name in a node.
- Nodes may contain an arbitrary number of key/value pairs, called properties:
  - Propery keys must be unique within any given node, and "should" contain only alpha-numeric characters or underscores, for compatibility with the widest variety of i/o formats. See section 4.3 for the general guidelines.
  - Property values may be of any Streamable Type (*not* pointers) which supports de/serialization via the standard C++ istream>> and ostream<< operators.

All that is basically saying is, the framework expects that data can be structured similarly to an XML DOM. Practice implies that the vast majority of data can be easily structured this way, or can at least be structured in a way which is easily convertable to a DOM. Whether it is an efficient model for a given data set is another question entirely, of course.

11.1.1 File extensions

File extensions are irrelevant for the library - client files may be named however you like. Clients are of course free to implement their own extention-to-format or extension-to-class conventions. (i tend to use the file extension .s11n, because that's really what the files are holding - data for the s11n framework.)

11.1.2 Indentation

Most Serializers indent their output to make it more readable for humans. Where appropriate they use hard tabs instead of spaces, to help reduce file sizes. There are plans for offering a toggle for indention, but where exactly this toggle should live is still under consideration. On large data sets indentation can make a significant difference in file size, and can account for up to 10% of a file's size for data sets containing lots of small data (e.g., integers).

11.1.3 Magic Cookies

This information is mainly of interest to parser writers and people who want to hand-edit serialized data.

Each Serializer has an associated "magic cookie" string, represented as the first line of an s11n data file. In the examples show in the following sections, the magic cookie is shown as the first line of the sample data. This string should be in the first line of a serialized file so the data readers can tell, without trying to parse the whole thing, which parser is associated with a file. The input parsers themselves do not use the cookie, but it is required by code which maps cookies to parsers. This is a crucial detail for loading data without having to know the data format in advance. (Tip: it uses `s11n::classload<SomeSerializableBaseType>()`).

Note that the i/o classes include this cookie in their output, so clients need not normally even know the cookie exists - they are mentioned here mainly for the benefit of those writing parsers, so they know how the framework will know to select their format's parser, or for those who wish to hand-edit s11n data files.

Be aware that s11n consumes the magic cookie while analyzing an input stream, so the input parsers do not get their cookie. This has one minor down-side - the same Serializers cannot easily support multiple cookies (e.g., different versions). However, it makes the streaming much simpler internally by avoiding the need to buffer the whole input stream before passing it on.

See `serializers.{h,cpp}` for samples of how to add new Serializers to the framework.

11.2 Overview of available Serializers

This section briefly describes the various data formats which the included Serializers support. The exact data format you use for a given project will depend on many factors. Clients are free to write their own i/o support, and need not depend on the interfaces provided with s11n.

Basic compatibility tests are run on the various de/serializers, and currently they all seem to be equally compatible for "normal" serialization needs (that is, the things i've used it for so far). Any known or potential problems with specific parsers are listed in their descriptions. No significant cross-format incompatibilities are known to exist.

11.2.1 funtxt (aka, SerialTree 1)

Serializer class: `s11n::io::funtxt_serializer`

This is a simple-grammared, text-based format which looks similar to conventional config files, but with some important differences to support deserialization of more complex data types.

This format was adopted from libFfunUtil, as it has been used in the QUB project since 2000, and should be read-compatible with that project's parser. It has a very long track record in the QUB project and can be recommended for a wide variety of standard data sets uses. It also has the benefit of being one of the most human-readable/editable of the formats (with parens being a close contender: section 11.2.4).

Known caveats/limitations:

- Known to have problems reading some unusual string contructs, such as properties which start with a quote but do not end with one.

Sample:

```
#SerialTree 1
nodename class=SomeClass {

     property_name property value
     prop2 property values can \
          span lines.
     # comment line.
     child_node class=AnotherClass {
     ...  properties ...
     }

}
```

Unlike most of the parsers, this one is rather picky about some of the control tokens[29]:

- Closing braces must be on a line by themselves.
- Each property must be on it's own line, but may span lines if each newline is backslash-escaped. Such newlines are retained when the data is read in.

This parser accepts some constructs which the original (libFunUtil) parser does not, such as C-style comment blocks, but those extensions are not documented because i prefer to maintain data compatibility with libFunUtil, and they play no role in the automated usage of the parser (they are useful for people who hand-edit the files, though).

11.2.2 funxml (aka, SerialTree XML)

Serializer class: `s11n::io::funxml_serializer`

The so-called funxml format is, like funtxt, adopted from libFunUtil and has a long track-record. This file format is highly recommended, primarily because of it's long history in the QUB project, and it easily handles a wide variety of complex data.

Known limitations/caveats:

- Does only very rudimentary character translation for XML entities - just enough for the input parser to reliably handle it. This will be fixed when problematic data actually shows up in a use-case.

[29] Hey, it was my first lexer - gimme a breat ;). Also, i wanted it to be compatible with libFunUtil's.

- To help support the various container serialization functions (section 7.4), this parser accepts node names which are numeric. That feature is not compatible with XML standards, and data files which use this feature may not be loadable by most XML tools without some filtering.
- Does not parse self-closing elements, e.g. `<node ... />`. Supporting that would make the output files more compact, so that's on the to-do list.

Sample:

```
<!DOCTYPE SerialTree>
<nodename class="SomeClass">

     <property_name>property value</property_name>
     <prop2>value</prop2>
     <empty></empty>

</nodename>
```

11.2.3 simplexml

Serializer class: `s11n::io::simplexml_serializer`

This simple XML dialect is similar to funxml, but stores nodes' properties as XML attributes instead of as elements. This leads to much smaller output but is not suitable for data which are too complex to be used as XML attributes.

This format handles XML CDATA as follows:

- Only CDATA wrapped in `<![CDATA[a block like this]]>` are recognized.
- At input-time all XML CDATA is stuffed into the "CDATA" property of the node.
- At output-time any data in a node's CDATA property is *not* saved as an XML attribute named "CDATA", but is instead stored as an XML CDATA block.

This is a non-standard extension to data node conventions, so clients which rely on this feature will be dependent on this specific Serializer. In practice, this feature has never been used client-side, but it seemed like an interesting thing to code at the time :/.

Known limitations:

- See the caveats/limitations notes in section 11.2.2. Most of those apply here.
- Not suitable for use with data which cannot be safely stored as XML attributes. That is, it is fine for storing numbers and other simple types, but storing complex strings may result in Grief (in the form of un-readable data).
- The XML attribute name "s11n_class" is reserved for use by the Serializer, to store the impl_class() of each node.

Sample:

```
<!DOCTYPE s11n::simplexml>
<nodename s11n_class="SomeClass"

     property_name="property value"
     prop2="&quot;quotes&quot; get translated"
     prop3="value">
     <![CDATA[ optional CDATA stuff ]]>
     <subnode s11n_class="Whatever" name="sub1" />
     <subnode s11n_class="Whatever" name="sub2" />

</nodename>
```

11.2.4 parens

Serializer class: `s11n::io::parens_serializer`

This serializer uses a compact lisp-like grammar which produces smaller files than the other Serializers in most contexts. It is arguably as easy to hand-edit as funtxt (section 11.2.1) and has some extra features specifically to help support hand-editing. It is arguably the best-suited of the available Serializers for simple data, like numbers and simple strings, because of it's grammatic compactness and human-readability.

Known limitations:

- Known to have problems with some unusual string contructs, such as properties which start with a quote but do not end with one.

Sample:

```
(s11n::parens)
nodename=(ClassName

     (property_name value may be a \("non-trivial"\) string.)
     (prop2 prop2)
     subnode=(SomeClass (some_property value))
     (* Comment block.

          subnode=(NodeClass (prop value))
          Comment blocks cannot be used in property values,
          but may be used in class blocks (outside of a property),
          "between" nodes, or in the global scope (outside the root node).

     *)

)
```

This format generally does not care about extraneous whitespaces. The exception is *property values,* where leading whitespace is removed but internal and trailing whitespace is kept intact.

When hand-editing, be sure that any closing parenthesis [some people call them braces] in propery values are backslash-escaped:

```
(prop_name contains a \) but that's okay as long as it's escaped.)
```

Opening parens may optionally be escaped: this is to help out Emacs, which gets out-of-sync in terms of indention and paren-matching when only the closing parens are escaped. When saving data the Serializer will escape both opening and closing parens.

11.2.5 compact (aka, 51191011)

Serializer class: `s11n::io::compact_serializer`

This Serializer read and writes a compact, almost-binary grammar. Despite it's name (and the initial expectations), it is not always the most compact of the formats. The internal "dumb numbers" nature of this Serializer, with very little context-dependency to screw things up while parsing, should make it suitable for just about any data.

Known limitations:

- Hand-editing it is very difficult. The data's sizes are encoded in the stream, preceeding the data, and any change in the data requires an update to the block's size - failing to do so effectively corrupts the data.
- Node names, impl_class names and property keys are limited to 255 characters in length.
- Property data is "limited" to 4GB per property.
- Input parser breaks with SOME single-letter class names, for reasons not yet fully understood.

Sample:

```
51191011[30]
f108somenode06NoClasse101a0003foo...
```

[30] "5119" is as close to "s11n" as i could get with integers. "1011" represents the data format version (there was a predecessor in 0.6.x and earlier).

11.3 Tricks

11.3.1 Using a specific Serializer

Simply pick the class you would and use it's de/serialize() member functions.

Normally you *must* select a class (i.e., file format) when saving, but loading is done transparently of the format.

See the various s11n::serialize<>() functions for a form which takes a SerializerType template argument.

11.3.2 Selecting a Serializer in s11nlite

See s11nlite::serializer_class() and s11nlite::create_serializer(), both of which take a classname for any registered subclass of s11nlite::serializer_base_type.

11.3.3 Multiplexing Serializers

This has never been done, but it seems marginally reasonable. i can personally see little benefit in doing so, however.

If you'd like, e.g., save to multiple output formats at once, or add debugging, accounting, or logging info to a Serializer, this is straightforward to do: create a Serializer. By subclassing an existing Serializer it is straightforward to add your own code and pass the call on. If you don't need s11n to see your Serializer, then don't write one, and simply provide a function which does the same thing.

Saving to multiple formats is only straightforward when the Serializer is passed a filename (as opposed to a stream). In this case it can simple invoke the Serializers it wishes, in order, sending the output to a different file. Packaging the output in the same output stream is only useful if this theoretical Serializer can also separate them later. If a multiplexer accepts an input stream, it must buffer the stream so that it can pass on the streamed data to each multiplexed Serializer, as the stream contents will be consumed by the first reader.

12 impl_class() & class_name<>: the whole truth

Once upon a time - the first few months of s11n's development - s11n developed a rather interesting trick for getting a type's *name* at runtime. Despite how straightforward this must sound, i promise ***it is not***. C++ offers no 100% reliable, in-language, understood way of getting something as seemingly trivial as a type's *frigging name*. While s11n's trick (shown soon) works, it has some limitations in terms of cases which it simply cannot catch - the end effect of which being that objects of BType end up getting the class name of their base-most type (e.g., "AType"). Let's not even think about using typeid for class names: typeid::name() ***officially*** *provides undefined behaviour*, which means *we won't even consider it.*

Historical note:

> Very early versions of s11n used a typeid-to-typename mapping, which worked quite well (and did not require consistent typeids across app sessions), but it turns out that typeid(T).name() *can return different values for T* when T is used different code contexts, e.g., in a DLL vs linked in to the main app. Thus that approach was, sadly, abandoned.

To be honest, the details of class names vis-a-vis s11n, in particular vis-a-vis s11n client-side code, are an amazingly long story. We're going to skip over significant amounts of background detail, theory, design philosophy, etc., and cut to the "hows" and the more significant "whys".

12.1 impl_class()

By s11n convention, impl_class() is a member function of Data Node types, used to get and set the string form of a type's name. For s11n this is significant at the following points:

1. When serializing an object, the node it is stored in should have it's impl_class() set to the object's class name. This is possible to achieve at the framework level for the majority of (all?) *monomorphic* types, but impossible to achieve *polymorphically* without some small amount of client-side work. In s11n this "small amount" of work comes in the form of setting a node's impl_name() to the string form of the Serializable's class' name. This is done in an object's serialize operator (not deserialize). If a type inherits Serializable behaviours it must set the impl_class() *after* calling the inherited behaviour, to avoid that the parent type overwrite the impl_class() of the subtype.
   Note that Serializable Proxies need to set the impl_class() to the name of the Serializable type, ***not*** to the name of the proxy type. Why? Read the next section and then it should be clear

2. When deserializing a node to a given BaseType, as in this code:
   `BaseType * b = s11nlite::deserialize<BaseType>( somenode );`
   s11n asks the BaseType's classloader (e.g., `s11n::classloader<BaseType>()`) for an object of type mapped to the name stored in `somenode.impl_class()`. The classloader, ideally, has a subtype of BaseType registered with that name (or it is BaseType's name, or maybe it can find the type via a DLL lookup). If so, the classloader will return a new instance of that type and s11n will hand off the data node to it using the internal API marshaling interfaces. If no class of the given name can be found by *BaseType's* classloader (other classloaders are not considered), deserialialization necessarily fails, as there is no object to deserialize the data into.
   When a data node is "directly" handed to a Serializable (e.g., `s11nlite::deserialize(srcnode,target)`) then the impl_class() is *irrelevant*, as s11n must assume that the given node and Serializable "belong together", semantically speaking. This property can be used to store arbitrary data in nodes, and have complementary de/serialize algorithms or functors which understand a common "data layout" within a given node. e.g., some of the various serialize[container]() variants use this: each pair of de/serialize functors supports one end of the data's "dialect", would be one way to put it.

In theory these points are all pretty straightforward, and all should make pretty clear sense. After all, to load a specific type it must have a lookup key of some type, and a classname makes a pretty darned convenient key type for a classloader. The classloader's core actually supports any key type, but s11n is restricted to strings, mainly for the point just mentioned, but also because non-strings aren't meaningful in the context of doing DLL searches for new Serializable types. Consider: what should an int key type be useful for in that context - interpretting it as an inode number? Thus, s11n internally uses only string-keyed classloaders.

Hopefully the significance of a node's impl_class() is now fully understood. If not, please suggest how we can improve the above text to make it as straightforward as possible to understand!

Side-notes:

- i do honestly believe it to be impossible in C++, using *only* in-language techniques, to 100% reliably get the class name for polymorphic types, not considering options like external (file-based) lookup tables. ***i would be extremely happy to be proven wrong!*** *Please* contact me if you know a magic trick for this!
- s11n actually did use external lookup tables for class names once, created by using the `nm` tool to extract all type names from an application/DLL *after* linking it. The immediate advantages are that it works fairly well, as it has access to all class names used in the binary (app/DLL), but it's cumbersome, build-wise, and *very* memory-hungry, as a huge number of the types in any binary are not at all relevant to the client for purposes of s11n (e.g., __gcc_blahblah_internal<Foo *,std::alocator<Foo> .....>).

12.2 classname<>(), class_name<>, name_type.h and friends

In the previous section i mentioned that s11n has a useful trick for getting the class name of a type. It's described in detail here...

To jump right in, here's how to map a type to a string class name. We'll show both ways, and soon you should understand why the second way is highly preferred. You do not need either of these if the class is registered via one of the core's registration supermacros, as those processes do this part already:

Method #1: (old-style: avoid this)

```
#include <s11n/class_name.h>
// ... declare or forward-declare MyType ...
CLASS_NAME(MyType);
```

Metod #2: (*highly preferred*)

```
#define NAME_TYPE mynamespace::MyType< TemplatizedType >
#define TYPE_NAME "mynamespace::MyType<TemplatizedType>"
#include <s11n/name_type.h>
```

By s11n convention, the class *name* should contain no spaces. This is not a strict requirement, but helps ensure that classnames are all treated consistently, which is critical if someone ever has to parse out a specific element of, e.g., a template type. That said, you can name the above type "fred" and it will work as well - just make sure not to use the same name for more than one type associated with the same classloader.

After the type is registered, the following code will return a (const char *) holding the type's name:

```
class_name<MyType>::name()
```

or it's convenience form:

::classname<MyType>()

Sounds pretty simple, right? If the preferred form is used, it *is* easy. If you use the macro form, you need to watch out for the following hiccups:

- MyType's name may not contain any commas: commas break C macros, as they are the argument delimiter character. A type with a comma in the name requires hand-specializing a class_name<T> or using name_type.h. Tip: see the `lib/cl/class_names` script for a code-generation shortcut.
- MyType should not (normally) be a typedef. *Aha!* You thought you'd use a typedef to get around the comma problem! Think again. When a typedef is passed to the macro, the typedef's name is registered as the class name. While this is not fundamentally evil (and *does* have valid uses!) it is generally not the desired behaviour.

The whole class_name<T> interface and conventions are covered in this list:

- `class_name<T>::name()` returns a (const char *) holding the value "T". Whether or not a return value of 0 is acceptable for unspecialized class_name<X>'s is still up for discussion. The current framework never returns 0, and returns an undefined string from non-specialized class_name<X> instantiations.
- In the case of `class_name<T*>`, pointer part of the type is *not* represented in the name. i.e., `class_name<T*>::name()` is "T". This behaviour has a long list of justifications. Suffice it to say that leaving it off simplifies significant parts of s11n's internals, and also makes s11n more flexible at the same time. e.g., it cuts the number of classloader registrations (and potentially factory objects) by half because we really don't need both "T" and "T*" for T - if we're classloading we're *always dealing with pointers*, so a descriptive string explaining that to the classloader is reduntant, maintenance-cumbersome, and ultimately unnecessary.
  Side note: interestingly, some of s11n algorithms can generically interpet e.g., "list<int>", as either "list<int>" or "list<int*>" (or even as pointers to one of those list types) with exactly the same algorithm: we let template code do all the type-juggling, using copy-based object creation for non-pointers and heap-based for pointer types. (It's pretty cool: see the sources for, e.g., s11n::list::list_serializer_proxy and s11n::map::pair_serializer_proxy.) For example, if you deserialize such a type to a list<int*> you will get a list of pointers to int, whereas if you pass it a list<int> as a container, that's exactly what it will convert the serialized data to.
  The point being: the removal of the "*" from "T*" is part of what makes such generic code easy to implement in s11n.
- `class_name<T>` lives in an *anonymous* namespace directly off of the global namespace. This *deceptively subtle* detail is *critical* for a number of reasons... all of them well out of scope here (no pun intended). Well, okay, let's summarize these rules:
  If class_name<T> lives in a non-anonymous namespace (i.e., named or global) then a binary in which class_name<T> was defined more than once will get ODR violations at link-time. *Anonymous namespaces* work around that problem - the specializations scattered throughout a source tree (potentially instantiated many times each) are collapsed at link-time into one instance of the class. As it is, class_loader<T> may not be specialized more than once for the same T *in the same compilation unit* (i.e., once per implementation file). Violating this rule will result in a compile-time error due to duplicate class definitions (i.e., a textbook example of an ODR violation). One implication of this is that putting the CLASS_NAME(MyType) in MyType.h is a guaranteed way to give all users of MyType a proper class_name<MyType> specialization.
  The anonymous namespace provides adequate flexibility on deciding where a class template specialization lives, to avoid many of the compile- and link-time problems associated with non-anonymous namespaces and utility classes such as this one, which tend to be used in lots of disparate places.
- `::classname<T>()` is is guaranteed to be *functionally identical* to class_name<T>::name(), and is provided because... well, because it's a lot easier to type and a lot friendlier looking. Note that sometimes you may be forced to fully qualify the call, e.g., ::classname<T>(), and it is generally preferable to do so, mainly for maintenance reasons (it makes the function easier to locate when you're not sure where it comes from).

The exact process of how class_name<T> or class_name<T*> get mapped to their string forms is undefined - it can happen in any way the specialization implementor wishes, as long as the specialization conforms to the above interface and are consistent: changes in the string form - even whitespace - may break older serialized data.

::classname<T>() will only return a valid value if a class_name<T> specialization exists (i.e., the above registration can been done), which means that any T passed to classname<T>() or class_name<T> must have an appropriate specialization if the class name is to be useful. Earlier versions of s11n aborted when an unspecialized class_name<T> was used, but this restriction has since been lifted.

13 SAM: {Serialization,s11n} API Marshaling layer

> **Achtung:** SAM is not Beginner's Stuff. This is, as Harald Schmidt puts it so well in a German coffee advertisement, *Chefsache* - intended for use by the "higher ups." This is *not* meant to discourage you from reading it, only to warn you that in s11nlite, and probably even when using the core directly, you will normally never need to know about SAM.

It's time to confess to having told a *little white lie*. Repeatedly, even willfully, *many* times over in this span of this document.

The Truth is:

> ***s11n's core doesn't actually implement it's own "Default Serializable Interface"!***

WTF? If s11n doesn't do it, who does?

Following computer science's oft-quoted "another layer of indirection" law, s11n puts several layers of indirection between the de/serialization API and... *itself*. To this end, s11n defines a minimal interface which describes only what the s11n core needs in order to effectively do it's work - no more, no less. s11n sends all de/serialize requests through this interface, which is generically known as SAM.

i admit it: i have, so far, *willfully* glossed *right* over SAM. However, i did so *purely* in the interest of keeping everyone's brains from immediately going all wahoonie-shaped when they first open up the s11n manual. As *you've* made this far in the manual, we can only assume that wahoonie-shaped suits your brain just fine. If that is indeed the case, keep reading to learn the Truth about SAM...

13.1 The SAM layer & interface

i've been telling you this *whole time* that types which support s11n's *Default Serializable Interface* are... well, "by default, they're already Serializables." In a sense, that's correct, but only in the sense that i've been "abstracting away" the very subtle, yet very powerful, features implied by the existance of SAM. Bear with me through these details, and then you'll surely understand why SAM is buried so far down in the manual.

At the heart of s11n, the core knows only about two small details:

- Data Node conventions (and only a small subset of them).
- SAM's two API functions and their conventions (which are identical to those of s11n's core de/serialize functions).

s11n's core doesn't know *anything* about *anyone's* de/serialize() interface *except* for that of SAM's. The core, to be honest, is essentially quite dumb - implemented in a relative *handful* of lines of code[31] - looking over the code now i'd guess that, if we don't count the two de/serialize_subnode() convenience funtions[32], it's *less than 30* actual code lines(!!!).

As with the rest of the framework, SAM is an *abstract concept, not a concrete type*. SAM itself, as a concept, defines only the interface between s11n's core and the world of client-side code. Versions 0.7.0-0.8.1 allowed clients to swap out the whole SAM layer, but this was removed in 0.8.2 because a) to save compilation time and object space by reducing frivilous class templates, and b) i honeslty don't think anyone will ever swap out the SAM. If someone *is* indeed interested in this contact us - it's trivial to re-implement without changing the client-side interface.

The following code reveals the *entire* client-to-core communication interface:

```
template <typename SerializableT>
struct s11n_api_marshaler {

    typedef SerializableT serializable_type;
    static const bool is_registered; // reserved for possible future use
    template <typename NodeType>
    static bool serialize( NodeType &dest, const serializable_type & src );
    template <typename NodeType>
    static bool deserialize( const NodeType & src, serializable_type & dest );

};
```

[31] Trivia: the majority of the code is split between the Serializers and, since only recently, de/serialization functors and algorithms.

[32] i never use them, anyway, as i find them somewhat out-of-place (but admittedly convenient), prefering to use s11n::create_child() instead.

By now that interface should look eerily familar. Note that static functions were chosen, instead of functor-style operator()s, based on the idea that these operations are activated very often, and i felt that avoiding the cost of such a frivilous functor was worth it. Additionally, this interface defines something "solid" for clients, as opposed to s11n's normal convention of using two functions with the same name - operator(). And (there's another, lamer reason) the operator()-style interface can easily generate ambiguity errors here, so it needs to be avoided.

Specializations of this type may define additional typedefs and such, but the interface shown above represents the core interface: extensions are completely optional, but reduction in interface is not allowed.

When a client makes an s11n call such as this one:

```
s11nlite::save( myobject, std::cout );
```

myobject will soon end up in the s11n core[33], as described in the next section.

It is important to understand *how* s11n "selects" a SAM specialization: by the *type* argument passed as a Serializable templatized type (be it a proxied POD, a MyType, or a proxied std::map - that's irrelevant). Thus, in the above call, s11n would use a SAM<myobject's type> specialization. We've jumped ahead just a tad, and it's now time to back up a step and, with the above in mind, get a better understanding of SAM's place in the s11n model...

13.2 SAM's place in the API calling chain

After client code initiates a de/serialization operation, once control gets to the s11n kernel the process goes something like this:

1. s11n asks the runtime environment for the default SAM for this BaseType? (i.e., the SAM<BaseType> specialization.)
2. s11n passes off the Serializable and Data Node to that marshaler.
3. At this point in the process the s11n kernel has *nothing* more to do except wait for a return value from SAM. [We could go on a 17-page tangent about exceptions at this point... but we won't.]
4. SAM is now in control of the request, and passed marshals the call into the API expected by it's parameterized type, which may be any API whatsoever. Normally this is the "direct" Serializable interface of the type or that of a proxy type.
5. SAM<BaseType> eventualy returns to the core, which then passes the results directly back to the user.

Note that in this context, "client code" might actualy refer to an algorithm or functor shipped with s11n - as far as the core is concerned, anything, including common "convenience" operations (e.g., child node creation) which happen before the the core calls, and while waiting on SAM, are "client code."

As a special case[34], SAM<X*> is single implementation, not intended to be further specialized - see below!

13.2.1 More about SAM<X*>

A single specialization does pointer-to-reference argument translation (since it's SerializableTypes will be pointer types) and blindly forwards them on to SAM<X>. Thus pointers and references to Serializables are internally handled the same way (where practical/possible), as far as he core API is concerned, and both X and (X*) can normally often used interchangeably for Serializable types passed to de/serialize operations.

The end effect is that if a client specializes SAM<*Y*>, calls made via SAM<*Y**> will end up at the expected place - the client-side specialization of SAM<*Y*>. See below for further information regarding this pointer-type specialization.

Client code SHOULD NOT implement any pointer-type specializations of s11n_api_translator<X*>[35]. If a client implements a SAM<X*> specialization the effects may range from no effect to a very difficult-to-track descrepency when *some* pointer types (e.g., X*) aren't passed around the same as others. Then again... maybe that's *exactly* the behaviour you need for type (SpecialT*)... so go right on ahead, just be aware of s11n's default handling of SAM<X*>, and the implications of implementing a pointer specialization for a SAM. Such tricks are not recommended, as it would be very difficult to track that down later, especially as the pointer/reference transparency of the API means you can't simply grep for the API being passed a dereferenced pointer.

[33] Sounds scary, doesn't it? Don't worry - typically these operations take only mircoseconds on modern computers, so he won't be in there long.

[34] Now that i re-read this, this is one of *extremely* few "special cases" in s11n - i have a special type of *non-love* for "special cases" in general, and avoid them in the interfaces at all costs.

[35] ... without much consideration, that is. There are conceivable uses for this, but they seem to be well beyond the realm of "common serialization needs", and thus we won't dwell on them here.

14 s11n-related utilities

This section list the utility scripts/applications which come with s11n.

14.1 s11nconvert

Sources: client/s11nconvert/src/main_dn.cpp

Installed as PREFIX/bin/s11nconvert

s11nconvert is a command-line tool to convert data files between the various formats s11n supports. This version *not* usage-compatible with version shipped with 0.6.x and earlier: please see the older documentation for that one's description.

Run it with -? or –help to see the full help.

Sample usages:

Re-serialize inputfile.s11n (regardless of it's format) using the "parens" serializer:

```
s11nconvert -f inputfile.s11n -s parens > outfile.s11n
```

Convert stdin to the "compact" format and save it to outfile, compressing it with bzip2 compression:

```
cat infile | s11nconvert -s compact -o outfile -bz
```

Note that gzip/bzip input/output compression is supported for *files*, but not when reading/writing from/to standard input/output[36]. You may, of course, use compatible 3rd-party tools, such as gzip and bzip2, to de/compress your s11n data.

15 Miscellaneous features and tricks

s11n has a number of features which may be useful in specific cases. While some of them require support code from "outside the s11nlite sandbox", a few of them are touched on here.

15.1 Saving non-Serializables

Let's say we've got a small main() routine with no support classes, but which uses some lists or maps. No problem - simply use the various free functions available for saving such types (e.g., section 7.4). This can be used, e.g., as a poor-man's config file:

```
typedef std::map<std::string,std::string> ConfigMap;
ConfigMap theConfig;
... populate it ...
// save it:
s11nlite::save( theConfig, "my.config" ); // also has an ostream overload
...
// load it:
s11nlite::node_type * node = s11nlite::load_node( "my.config" ); // or istream overload
if ( ! node ) { ... error ... }
s11n::map::deserialize_streamable_map( *node, theConfig );
delete( node );
// theConfig is now populated
```

Alternately, simply use a s11n::data_node as a primitive config object.

[36] Sorry, i don't have a compress-in-memory streambuffer for these.

15.2 "casting" Serializables with s11n_cast()

Serializable containers of "approximately compatible" types can easily be "cast" to one another, e.g., list<int> and vector<int>, or even list<int> to vector<double*>.

The following code will convert a list to a vector, as long as the types contained in the list can be converted (by C++) to the appopriate type:

```
bool worked = s11nlite::s11n_cast( mylist, myvector );
```

Done!

Reminder: if this fails then myvector may be partially populated. If it contains pointers it may need to be cleaned up - see s11n::free_list_entries() for a convenience function which does that for arbitrary list types.

15.3 Cloning Serializables

Generic cloning of any Serializable:

```
SerializableT * obj = s11nlite::clone<SerializableT>( someserializable );
```

As you probably guessed, this performs a clone operation based on serialization. The copy is a polymorphic copy insofar as the de/serialization operations provide polymorphic behaviour. Reminder: make sure to use the proper (i.e., base-most) SerializableT type for the template parameter.

15.4 zlib & bz2lib support

s11n supports file de/compression using zlib and bz2lib if `configure` finds the appropriate libraries and headers. However, in the interest of data file portability/reusability, *file compression is off by default.* Use `s11n::compression_policy()` to set the library's default file compression policy (defined in `file_utils.h`).

All functions in s11n's API which deal with input files transparently handle compressed input files if the compressor is supported by the underlying framework, regardless of the policy set in `s11n::compression_policy()`: see s11n::get_istream() and get_ostream() if you'd like your client-side code to do the same. Note that compression is not supported for arbitrary *streams*, only for *files*. Sorry about that - we don't have full-fledged de/compressor streambuffer implementations, only file-based ones (if you want to write one, PLEASE DO! :).

As a general rule, gzip will compress most s11n data approximately 60-90%, and bzip often much better, but bzip takes 50-100% more time than gzip to compress the same data. The speed difference between using gzip and no compression is normally negligible, but bzip is *noticably* slower on medium-large data sets.

To *completely* disable gzip/bzip de/compression in your libs11n installation, run:

```
./configure --without-zlib --without-bzlib [any other args, like --prefix=...]
```

If you don't use the supplied build tree, to disable compression support you should define these C macros, ideally in config.h or the global compiler options (or similar):

```
HAVE_ZLIB=0
HAVE_BZLIB=0
```

And remove gzstream.* and bzstream.* from your project file(s).

There is no benefit whatsoever in disabling such support, but hey... it's your source tree.

As a final tip, you can enable output compression pre-main(), in case you don't want to muddle your main() with it, using something like the following in global/namespace-scope code:

```
int bogus_placeholder = (s11n::compression_policy( s11n::GZipCompression ),0);
```

That simply performs the call when the placeholder var is initialized (pre-main()).

Trivia note: this trick is actually the same one the classloader uses to register classes: they send their registration to the classloader when the app or DLL they are in goes through the static-data-init phase, i.e. when opened by the OS.

15.5 Using multiple data formats (Serializers)

It is possible, and easy, to use multiple Serializers, from within in one application.

Traditionally, loading nodes without knowing which data format they are in can be considerably more work than working with a known format. Fortunately, s11n handles these gory details for the client: it loads an appropriate file handler based on the content of a file. (Tip: clients can easily plug in their own Serializers.)

Saving data to a stream necessarily requires that the user specify a format - that is, client code must explicitely select it's desired Serializer. Once again, s11nlite abstracts a detail away from the client: it uses a single Serializer by default, so s11nlite's stream-related functions do not ask for this.

Data can always be converted between formats programmaticaly by using the appropriate Serializer classes, or by using the s11nconvert tool (see section 14.1).

It is not possible, without lots of work on the client's side, to use multiple data formats in *one data file* - all data files must be processable by a single Serializer.

15.6 Loading Serializables dynamically via DLLs

s11n's default classloader is DLL-aware. When it cannot find a built-in class of a given name it looks for the file ClassName.so in a configurable search path available via cllite::class_path(). The DLL loading support is fairly easy to extend if the default behaviour is too simplistic for your needs, but it's customization is, so far, undocumented: see lib/cl/src/cllite.h.

15.7 Renaming the s11n namespace

Largely in the interest of making s11n more viable for direct inclusion into other projects' trees, the supplied build tree supports this rather eccentric feature:

```
./configure –s11n-namespace=myns
```

That changes *all* of the s11n-namespaced source code to use the given namespace. The new namespace must be top-level, without any :: parts (sorry for that limitation, but the code maintenance effort involved in a variable-depth-ns solution is impractical). This name-space change is set at configure-type and applied at build-time: the "real sources" have placeholder tokens which get filtered into the namespace during the build.

Before you do this, be aware that it has *far-reaching* implications, some of which are:

- #include <s11n/foo.h> becomes #include <myns/foo.h>
- headers are installed to PREFIX/include/myns
- The name of the compiled library file is changed:
  libs11n.{so,a} become libmyns_s11n.{so,a}
- All macros which are prefixed with "s11n_" are changed to be prefixed with "myns_". e.g., s11n_CLASSLOADER_REGIS
  becomes myns_CLASSLOADER_REGISTER1.
- The like-named, like-signatured classes myns::Foo and s11n::Foo are actually *completely different, unrelated types*, which may have major implications when writing to the interface of one or the other. For example, some functions may need to be made into template functions to be able to handle both (basically identical) interfaces. This also means that the two Foo types use two completely different classloaders.
- Documentation (like this file) is *not* updated to contain the new namespace.

Some of these changes are applied to allow differently-namespaced versions of the lib to co-exist on the same systems, in both source and library forms, without collisions.

Given the wide-reaching effects, clients should think not once, not twice, but thrice before actually using this feature. Again, it is mainly provided in the interest of people who want to copy/paste the s11n tree directly into their project's tree and use their own namespace for the s11n framework.

15.8 Sharing Serializable data via the system clipboard

Experience has shown that holding pointers to objects in the system clipboard can be fatal to an application (at least in Qt: if the object is deleted while the clipboard is looking at it, the clipboard client can easily step on a dangling pointer and die die die). One perhaps-not-immediately-obvious use for s11n is for storing serialized objects in the clipboard as text (e.g. XML). Since nodes can be serialized to any stream it is trivial to convert them to strings (via `std::ostringstream`). Likewise, deserialization can be done from an input string (via `std::istringstream`). It is definately not the most efficient approach to cut/copy/paste, but it has worked very well for us in the QUB project for several years now.

Additionally, QUB uses XML for drag/drop copying so if the drag goes to a different client, the client will have an XML object to deal with. This allows it, for example, to drop it's objects onto a KDE desktop.

Assuming you serialize to a common data format (i.e., XML), this approach may make your data available to a wide variety of third-party apps via common copy/paste operations.

15.9 s11n and toc: "the other ./configure"

s11n is co-developed with another pet-project of mine, a build environment framework for GNU systems called toc:

http://toc.sourceforge.net/

In the off-chance that you just happen to use toc to build client code for s11n, see toc/tests/libs11n.sh for a toc test which checks for libs11n and sets up the configure/Makefile vars needed to compile/link against it.

16 Caveats, gotchas and some things worth knowing

16.1 Serializing class templates

All in all, serializing class templates is implemented just like all other classes. There is one especially tricky element, however: given that we don't know in advance what parameterized types will be used, how do we set the proper type *name* (i.e., the target data node's impl_class())?

One approach - the only one i've found, for that matter - is to use a class_name<X> *partial template specialization*. See section 12 for information on class_name<>, and see s11n's sam_standard_containers.h for examples of implementing the appropriate partial template specializations for class_name<>. That header contains, for example, the specializations used for getting std::list/vector/map class names.

16.2 Compiling and linking s11n client applications

Use the s11n-config script, installed under PREFIX/bin, to get information about your libs11n installation, including compiler and linker flags clients should use when building with s11n. It may (or may not) be interesting to know that s11n-config is created by the configure process.

As with all Unix binaries which link to dynamically-loaded libraries, clients of libs11n must be able to find the library. On most Unix-like systems this is accomplished by adding the directory containing the libs to the LD_LIBRARY_PATH environment variable. Alternately, many systems store these paths in /etc/ld.so.conf (but editing this requires root access). To see if your client binary can find libs11n, type the following from a shell:

ldd /path/to/my/app

Example output:

```
stephan@ludo:~/cvs/s11n/client/sample> ldd ./test
    libltdl.so.3 => /usr/lib/libltdl.so.3 (0x40034000)
    libs11n.so.0 => /home/stephan/cvs/s11n/lib/libs11n.so.0 (0x4003b000)
    libz.so.1 => /lib/libz.so.1 (0x400b7000)
    libbz2.so.1 => /usr/lib/libbz2.so.1 (0x400c6000)
    libstdc++.so.5 => /usr/lib/libstdc++.so.5 (0x400d7000)
    libm.so.6 => /lib/i686/libm.so.6 (0x40197000)
    libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x401ba000)
    libc.so.6 => /lib/i686/libc.so.6 (0x401c2000)
    libdl.so.2 => /lib/libdl.so.2 (0x402f5000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

16.3 Thread Safety

To be perfectly correct, there are no guarantees. i have no practical experience coding in MT environments.

The s11n code "should" be "fairly" thread-safe, with one *known* major exception:

Some of the lex-based input parsers are known to be 100% thread-unsafe (or un-thread-safe, if you prefer):

- compact_serializer
- simplexml_serializer

The others (parens, funtxt and funxml) have been extensively reworked to use instance-specific internal buffers, as opposed to global data, and are believed to be thread-safe.

The lack of thread safety guarantees means that s11n cannot currently be safely used in most network communication contexts, for example, as they would presumably want to read from multiple client-server streams.

The guilty code is probably almost all in the flexers, though some of the shared objects (e.g., classloaders) could conceivably be affected (but probably not enough to make any practical difference, at least in the case of the classloaders).

16.4 Object Ownership vis-a-vis Serialization

(Many, *many* thanks to Marshall Cline, of C++ FAQ fame, for his feedback on this!)

It is important to keep in mind that s11n does not inherently manage *any* object relationships. Instead, it leaves this task to the client, who will presumably manage them via serialization operators or via algorithms. To give an example, the core does not know anything about de/serializing a std::map<X,Y> - it is up to the client to serialize the map. It just so happens, however, that the library comes with some algorithms for doing this.

This library essentially takes the same approach as one does when managing pointer ownership. To be clear: that has no inherent relationship to serialization, except that the two are conceptually similar. To clarify what is meant by this we will use a simple example which every C++ developer has certainly come across:

When dynamically allocating objects, it is always important to determine how they will be destroyed. More specifically, it is important to determine who will destroy them. Quite often - probably most of the time - the object which allocates the memory is also the one to free it. Sometimes a smart pointer is the one to manage this, and sometimes pointer ownership is passed off to objects other than the one which allocated it (perhaps to client code). This library takes a similar approach to managing de/serialization of objects. Thus, clients must decide where a given object will be de/serialized. Oftentimes this is handled in a parent object's serialization operators or via an algorithm designed to manage a specific type of parent-child relationship. To go back to the example of map<X,Y>: s11n::map::serialize_map() can manage the serialize-time relationships of a collection of (Y*) to their parent object, a map<X,Y*>. Conversely, s11n::map::deserialize_map() manages those relationships at deserialize-time. The serialization relationship of the Y pointers to their container may or may not be equivalent to their memory or parent ownership, but is handled in a conceptually similar way. That is to say that each (Y*) has a well-defined "serialization owner" - the s11n::map::de/serialize() algorithms.

Thus when we speak of "serialization ownership", we are speaking of a process which is conceptually similar to "pointer ownership." More specifically, we are speaking of the code which is responsible for de/serializing a given object. While it is very possible that pointer/memory ownership of a given object are managed by the same code which owns serialization, there is no specific rule which says this should be the case.

A data structure containing objects A and B, which both serialize each other, will cause infinite recursion in the s11n core during serialization unless one or both of those structures can accomodate the recursive relationship vis-a-vis serialization. Such recursion is presumably indicative of mis-understood or incorrect *serialization ownership*. Consider: presumably only an object's serialization owner should serialize that object, and child objects should generally never have more that serialization owner. Data Node-based de/serialization (as opposed to Serializable-based) never infinitely recurses because those structures simply don't manage the types of relationships which can lead to cycles. In other words, any such recursion must be coming from client-manipulated Data Node trees. (As Marshall has pointed out: a tree is by definition acyclic, and thus once there are cycles it is no longer a tree.)

One advantage to this "s11n doesn't know anything" approach, as opposed to the library blindly serializing all objects it finds, is that clients can customize the de/serialization handling for any given structure to fit their needs. For example, serialize_map() does not do what the client wants, another algorithm can be dropped in to replace it for a given case. By adding a serialization proxy, this algorithm can be transparently plugged in to the framework, such that users of a special-case map need not even know they are using a customized algorithm.

16.5 Cyclic data structures

Can s11n handle cyclic data structures?

The short answer is: yes

The longer answer is: there are currently no algorithms shipped with the library which inherently handle cycles. Thus clients must write their own.

17 Common problems

In this section i impart some of my hard-earned knowledge with the hope that it saves some grey hairs in other developers...

17.1 Satan speaks through the console during compilation

If, during compilation, your terminal is filled with what appear to be endless screens of gibberish from the mouth of Satan himself, don't panic: that's the STL's way of telling you it is pissed off.

It may very well be one of these common mistakes (i do them all the time, if it's any consolation):

- You're trying to serialize a type which isn't yet registered with s11n. This often happens when serializing containers: remember that the contained type(s) must be Serializables, and that a map's value_type (a pair type) must also be made Serializable in order to make a map Serializable.
- You've swapped the arguments for a de/serialize() call. By convention, nodes always come before Serializables in the parameter list. Swapping these will cause you no end of error messages from Hell, with things like, "no such function ... list<..>::impl_class()..." or "list<..>::children()". The first hint that the args are swapped is that it's trying to call a Data Node API function on your Serializable.
- You've tried to pass a pointer as a node argument. Serializables are generally accepted regardless of whether they are passed as pointers or not, but nodes are only passed by reference. Why? Because nodes are easy for the API to control in this regard and Serializables aren't, so they get some extra leeway (besides, it was easy to implement the pointer-to-reference translation at one point in the core). This property internally simplifies many operations on Serializables, as well.
- You have jumped from s11nlite to s11n without being aware of the different template args required by like-named functions in the s11n namespace. Shame on you. Almost without exception, the s11nlite:: functions with the same name as s11n:: functions are missing one template parameter (the first one) - the data node type - because s11nlite abstracts that detail away. That said, in many cases the calls are identical, because template type resolution will do the right thing, in which case the s11n/lite functions are basically the same (lite duplicates/forwards lots of functions simply to keep a whole usable client-side API in that namespace). Be sure to check for differences, though, before freely switching between the two (see the API docs).
- Const errors during a de/serialize call: make sure that your Serializable's [proxy's] serialization operators have the proper constness, as defined in section 4. In the case of a proxy, you may have to split it into two functors: one each for de/serialization, and be sure to add S11N_DESERIALIZE_FUNCTOR to the registration call. This shouldn't normally be necessary, however.
- When fetching a child node during a deserialize operation using, e.g., s11n::find_child_by_name(), be sure you use a (const NodeType *) and not a non-const (NodeType *), as the parent object is const in that context.
- When iterating over containers, be sure to use const_iterators if the NodeType or SerializableType passed to the function are const, as appropriate.

To be honest, though, those are just the common ones - any minor violation in usage will cause the STL to go haywire, as i'm certain you have already experienced many times in your coding life.

17.2 Containers serialize, but fail to deserialize

This is almost invariably caused by a simple logic error:

(Been there, done that.)

When serializing containers, it is essential that each container is serialized into a separate node. After all, each container is ONE object, and one node represents one object . It is easy to accidentally, e.g., serialize both a list<int> and map<string,string> into the same node.

If you've done that, there may be two ways to recover from it (assuming you need to recover the data):

- Edit the output file and split the nodes up. Feasibility depends on the Serializer used: some may not be hand-editable. (Tip: s11nconvert can convert it for you - section 14.1.)

- Programmatically fish the data out of the node, e.g., using s11n::find_children_by_name() to separate the various children. In a worst-case (all entries have the same name) you'll need to do it based on impl_class(), but that would be no fun at all, as they are unpredictable. (Expecting an "AType" node? Think again - you got a "BType"!)

Also, it is essential that you use always use complementary de/serialization algorithms/proxies. For example, if you use serialize_streamable_map() to save a map, then use *only* deserialize_streamable_map() to deserialize it, as any other algorithm may structure the serialized data however it likes, as defined in it's documentation. Be aware of each algorithm's weaknesses and strengths before settling on it, because changing later may not be feasilbe (old data won't be readable without, e.g., special-case code to check for it and use the "old" algorithm).

17.3 ::classname<T>(), name_class.h and friends

As of 15 March 2004 [will soon be s11n 0.8.0] the CLASS_NAME() macro is fully obsoleted by the name_class.h "supermacro", which can support types with commas in their names (any type name is valid). The underlying mechanics of them are identical - they are compatible, but the CLASS_NAME() macro cannot be used in all cases, as described later, and may eventually be phased out.

17.3.1 Duplicate definitions of class_name<T>

This can be caused by at least these things:

- Using CLASS_NAME(X) (or a variant of it) more than once for the same type in the same compilation unit.
- Calling class_name<X> (or ::classname<X>()) before class_name<X> is actually *declared* (not necessarily defined). This normally happens when, e.g., X calls ::classname<X>() to get it's own name, e.g., in it's serialize operator. When this is called in template code it is not as much of a problem, because the call is not complete analysed until the template code containing it is instantiated (i.e., called for the first time).

In the second case: if you want to use ::classname<X>() from within X's code in there are workarounds, but they're not necessarily pretty:

- Before defining the class, forward-declare it your class, then use CLASS_NAME(X) (or equivalent).
- Make sure you call it from inside a member function template, then call CLASS_NAME(X) directly after declaring class X.
- If X's declaration and implementation are separated (not always possible with template code) the answer is trivial: simply register the name from X.cpp after including X.h. Note, however, that if you do NOT call class_name<X> from within that impl file (or from something included by the impl file), then class_name<X> *is never actually instantiated* and will not be linked in with your code. The end result will be a very confusing "undefined reference to ...class_name<X>..." error. ("But I DID define it RIGHT HERE, you f*!#&@ compiler!!!!" Indeed, you may have, but it is never actually created until used once in the same compilation unit. This is a normal feature of templates.)

17.3.2 class_name<T> not defined

(Also see the previous section.)

Compile-time:

The most common cause is that CLASS_NAME(T) has not be called before class_name<T> is used. This one is normally easy to fix. It is really easy to forget to define a class_name<T> for arbitrary template-typed T's, by the way, but there is no known way to programmitically get their names without a helper like class_name<T>.

Link-time:

There is a more complex case i hit once which took me ***hours*** to track down:

If a class_name<T> specialization is defined in an implementation file, but is never used (instantiated) within that impl file, then class_name<T> is *never actually instantiated*. Thus, code outside of that impl file which call class_name<T> does not see the macro-generated code from the original impl file, as it was never actually instantiated by the compiler.

17.3.3 map<X,Y>::value_type vs pair<X,Y>

map<X,Y>::value_type is not pair<X,Y>, but pair<const X,Y>.

Thus, class_name<pair<X,Y> > will return a different value than class_name<mymap::value_type>, because template type resolution is sending you to two completely different class templates. After discovering this, a class_name<const T> specialization was put in place to try to avoid this, so it "shouldn't happen" again.

In any case, this may also cause a problem when proxying maps via s11n. The map's value_type type must also be proxied (tip: pair_serializer_proxy). When proxying a map/pair combination, you should register map's value_type typedef instead of pair<X,Y>. That said, most maps do not require any special registration or proxy declaration, as they are handled by s11n::map::map_serializer_proxy, which handles this const/non-const descrepancy.

17.4 Abstract base types for Serializables

s11n can handle abstract base types: simply add this line before including the registration supermacro, reg_serializable.h:

```
#define S11N_ABSTRACT_BASE
```

That's all. This does not have to be added for subclasses of that type. As usual, this macro will be unset after including the supermacro.

For the curious: this installs a no-op object factory for the type, as those types cannot be instantiated, and thus cannot be created using new(). As far as the classloader is concerned, trying to instantiate a registered abstract type simply causes 0 to be returned.

18 Where to go from here?

Since the new s11n framework is *so* new there is very little sample client-side code for it. There are a couple client-side apps code in the s11n source tree, which will certainly prove informative to those starting out with s11n:

client/sample/src/demo_struct.cpp

client/sample/src/demo_hierarchy.cpp

client/s11nconvert/src/main.cpp

The web site is updated fairly often, and you just might find something interesting over on there if you check back once in a while:

http://s11n.net

.....

As always:

- The source tree is *always* the most-definitive source of information, but the web site is also updated fairly often, sometimes a bit in advance of upcoming changes.
- i am always open to getting mails with questions about s11n, so don't hesitate to ask. i will ask that you please browse the manual first, but i certainly do *not* expect you to scour every web page or code file before posing a question. i understand that the documentation has some gaping holes in it right now, and i will be happy to fill those holes by directly answering your questions.

Once again: thats a lot for taking the time to consider adding s11n to your toolkit!

— stephan@s11n.net

19 Index

DAMN: turning on the index breaks LyX's exports! And the index doesn't show up in the lyxport-converted versions...

Sheesh...