s11n
an Object Serialization Framework for C++

Version 1.1.x

s11n-devel@lists.sourceforge.net - http://s11n.net

May 29, 2005

Abstract

This document describes s11n (and "s11nlite"), an object serialization framework for C++, version 1.1.x "development/experimental" (which will someday become the 1.2.x "stable" tree). 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!

ACHTUNG #1: the HTML version of this document is KNOWN TO HAVE ERRORS introduced by the LyX-to-HTML conversion process, such as arbitrarily missing text. Please consider reading a LyX or PDF copy instead of an HTML copy. HTML versions are released primarily as a convenience for web-crawling robots, not all of which can read PDF.

ACHTUNG #2: this is a "live" document covering an in-development software library. Ergo... it may very well contain some misleading or blatantly incorrect information! Please help us improve the documentation by sumbitting your suggestions to our mailing list!

ACHTUNG #3: this doc is currently a bastard mix of s11n 1.0 and 1.1/1.2, and will likely remain so until the 1.2 release draws near (rough current estimate is late 2005). Thus it contains a lot of info which is true for 1.0 but not 1.1 *and vice versa*. The basic architectures and conventions are 98% identical, but they are structured physically differently and have some relatively minor usage differences.

Document CVS version info:
$Id: s11n.lyx,v 1.2 2005/05/25 18:16:12 sgbeal Exp $
Maintainer: stephan@s11n.net (list: s11n-devel@lists.sourceforge.net)

Contents

1 Preliminaries

1.1 License

The library described herein, and this documentation, are released into the Public Domain. Some exceptional library code may fall 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, in which case it is Public Domain, 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, and in absolutely no cases does this project use licenses which modify the license of code linked against it.

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.

Whatever the license, however, i will request that if you redistribute your own libraries based off of this code, please do not use the same *installed* binary/library/header filenames. For example, if you redistribute libs11n, please do not install the library as libs11n.so, nor the headers under <s11n.net/s11n/...>. Doing so will inherently compliciate cases where both of our copies of s11n are used on the same systems.

1.2 Disclaimers

1. This manual will make *no sense whatsoever* to most people. It is target at experienced C++ programmers (”intermediate level” and higher), 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 continually under development and is constantly being tweaked. The basic model it is based on 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. Should you feel compelled to contribute financially to this project, please do so via the donation program hosted by SourceForge and PayPal:

`https://sourceforge.net/donate/index.php?group_id=104450`

Donations will go toward keeping the web site online and the domain name registered, and potentially to cover internet access fees. If anyone is interested in providing a grant to this project, please contact us directly. We would be thrilled.

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 few who know me would say that i am a pedantic person, i am extremely pedantic when it comes to documenting software: if you find any errors or gaping holes in these docs, please point them out!

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 0.7 framework can do this with "relative" ease, and 0.8+ makes it trivial in many cases.
- 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.
- s11nlite was developed largely because of Ton's and Gary's feedback.
- The massive build tree re-orgs between 0.8.x and 0.9.x were inspired by the Debian Project's **martin f. krafft** (yes, he prefers it spelled lower-case).

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 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 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! :)

1.4 Credits

There is no one, complete list of all people who have influenced this project. A partial list, in no particular order:

(If i have left *you* off of the list, please let me know!)

- My mother and step-father **Bonnie & David Pickartz**, my father and step-mother **Joseph & Gail Hudgins**, my step-mother-by-adoption **Kathy Beal**, and my belated adopted dad, **Gerry Beal**[1]. They just all need to be thanked in general. The 0.9.x branch, from 0.9.4 to 1.0.0, was directly funded by ***very gracious*** donation from the Pickartz family.
- **Rusty "Bozo" Ballinger** wrote the conceptual forefather of s11n (http://libfunutil.sourceforge.net). (Rusty, if you're still out there, *get in touch!*) As far as i know, Rusty also coined the phrase "s11n" as a short form of "serialization", which i then stole as a domain name.
- **Ton Oguara** accidentally inspired the whole 0.6 –> 0.7 rewrite/refactor by showing me how much client-side effort was *really* needed to de/serialize class templates.
- **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. Gary also is credited with coming up with a useful naming convention for Serialization Proxies, `MyType_s11n` (a convention i use in many of my project).
- **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. It is still used to this day in registering class factories.
- **martin f. krafft**, of the Debian Project, put in a great deal of effort to get the 0.8.x series into the Debian, and was the driving force behind the 0.8.x –> 0.9.x source tree re-orgs. His continued feedback is always insightful.
- **Marshall Cline**, of C++ FAQ fame, helped to correct some of the errors in the documentation regarding cycles, joins and trees. His FAQ has a great section on the topic of serialization in C++: `http://www.parashift.com/c++-faq-lite/`
- **Robert Ramey**, author of the Boost serialization library (`http://www.rrsd.com`), for several insightful email conversations on the topic of serialization. His well-crafted library is compared to this one (we might even say praised) in some detail in section 23.
- **Steve Madere** suggested adding unit tests to the source tree, and within an hour of doing so 2 significant bugs were caught and fixed. He also made a financial contribution via the SourceForge/PayPal donation system.
- **Andreas Jochens** provided several patches for compiling the 0.8.x tree under gcc 3.4.
- **Mike Radford** provided more patches for gcc 3.4 and gave me ssh to his box to let me fix a couple more.
- **Patrick Lin** demonstrated and helped localize a long-standing container-of-containers deserialization bug-in-waiting which couldn't wait any longer on his machine (existed from 0.8.x until 0.9.17).
- **Keven Weber** helped track down a couple bugs by allowing me ssh access into his machine, where the bugs were appearing.
- **Christian Prochnow**, project lead of P::Classes (http://pclasses.com), allows me to integrate s11n support into P::Classes 2.x. The port provides a great opportunity for bug finding and cleanups.
- **Dr. Marc Duerner**, first for inviting me to pclasses.com and secondly for his continued and on-going feedback and hacking sessions.
- **Gregor Jehle**, also of pclasses.com, reported compile problems on AMD64 and allowed me ssh access to his box to track down and fix them.
- "**Ashran**", author of the *Hackersquest* Everquest(tm) emulator (`http://hackersquest.org`), often puts up with me ranting (like a madman, i might add) about the latest s11n breakthroughs.
- **Peter "What's Happ'nin'!!?!!?!?!" Angerani**, my long-time friend and mentor, for his continued support and feedback.

[1] i've got 8 brothers and 4 sisters. Yes, i actually *do* know all of their names: (in no particular order) Toby, Gerald, Ty, Trevor, Teven, Wayne, Wesley, David, Margorie, Melisa, Ashley and Cindy (though i've never actually met Cindy). Their birthdays? Err.... ???

- To my esteemed Unix-loving colleagues, **Ralf Lehmann** and **Martin Tessun**, for agreeing, after bribing them with their own personal Easter Egg in this document, to look over this manual for me. (*Hier ist euer Easter Egg, Jungs!!!*)
- **SourceForge** (`http://sourceforge.net`) has been hosting my code since 2000, and without them s11n would have neither mailing lists, a bug tracking site, nor a public CVS tree. i encourage all users of SourceForge to support their service by buying a yearly subscription to their site.

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

- **Scott Meyers** - a *huge* percentage of my code is influenced by Scott's always-practical advice. All of his books must be on any C++ coder's bookshelf. *Here's your biggest fan, Scott!*
- **Andrei Alexandrescu** - his *Modern C++ Design* was the necessary catalyst i needed for realizing the classloader implementation, and provided the basis for the internals of the `phoenix::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 member-based interfaces 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. ;) If i recall correctly, Stephen also introduced me to the idea of Monostates, which are conceptually similar to what i've been calling "Context Singletons."
- *C++ Templates: The Complete Guide*, by **Nicolai M. Josuttis** and **David Vandevoorde**, as well as Josuttis' *The C++ Standard Library*, were instrumental in implementing much of the template code used by this library. The latter is always the first book i reach for when i've got a question about the STL, and 99% of the time it has the answer[2].

2 Introduction

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

What?!?! You've got a:

```
std::map< int, std::list< std::map< double, FooObject<X *> * > >
```
[4]

?!?!?

Null problemo, amigo:

s11n is here to ***Save Our 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*

[2] i say 99% because i generally mistrust statements which include a "100%" qualifier, but the truth is i can't remember a time when this book didn't have what i was looking for.

[3] Plain Old Data types, such as `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. Interestingly, for many cases (those not needing a classloader) we can actually get by without knowing the type's name.

[5] "s11n" 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" - the number represents the number of letters removed from the middle of the word.

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.

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), plus it tries to fill the detail gap left in the API documentation. 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 info into this document and API docs.

Between this manual, the API documentation, and the s11n web site, pretty much all of your questions about the library should be answered. If not, feel free to email us.

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

2.2 s11n's Dream

Anyone who has had to hand-code save and load support for their data, even if only for relatively trivial containers and data types (e.g. even non-trivial strings), will almost certainly agree with the following statement:

> 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[7]. 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++...

In the 1980's IOStreams, the predecessor of the current STL iostreams architecture, brought us, the C/C++ development community, *tremendous* steps forward, compared to the days of reading data using classical brute-force techniques, such as those provided by standard C libraries[8]. That model has evolved further and further, and is now an instrumental part of almost any C++ code[9]. However, the practice of directly manipulating data via streams is showing its age. Such an approach is, more often than not, not suitable for use with the common higher-level abstractions developers have come to work with over the past decade (for example, what does it *really* mean, semantically speaking, to send a UI widget to an output stream?).

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[10], leapt into the limelite. Pratically overnight, XML evolved into *the* generic platform for data exchange and, perhaps even more significantly, *data conversion.* 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. *These points 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...

[6] i'd be verry happy to get a publishing offer! :)

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

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

[9] 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.

[10] [Standard,Structured] Generic Markup Language

We're in the 21st century now. In languages like Java(tm) and C# serialization operations are basically built-in[11]. 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 C++ 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's 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 (and specializations), Meyers, Alexandrescu, Strousup, Sutter, Dewhurst, PHP, "Gamma, et al", comp.lang.c++, application frameworks, Java[12], and... even lowly ol' me (yeah, i'm the poor bastard who's been pursuing this problem for 3+ years ;), and attempts to create a unified, generic framework for saving... well, damned near anything. Actually, *saving* data is the easy part, so we've gone ahead and thrown in *loading* support as an added bonus ;).

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"[13], and it's capabilities *far, far* exceed any original plans i had for it. Truth be told, i use it in *all* of my C++ code. i can finally... *finally*, **FINALLY *SAVE MY OBJECTS!!!!***

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

> ***It's about damned time!!!***

2.3 Main features

For the most part, the features list is the same as for s11n 1.0.x. For those of you who haven't used 1.0.x, 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.*[14]
- Provides client code with *easy* de/serialization of arbitrary streamable types and user-defined Serializable types.
- Out of the box it supports all standard STL containers: `std::list`, `vector`, `set`, `multiset`, `map`, `multimap` and `valarray`.[15]
- 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.

[11] Though i do have very deep fundamental differences with Java's built-in serialization model!

[12] 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.

[13] 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 the classloader.

[14] On a features/technical level, the only currently-existing C++ serialization framework which can even begin to compare with s11n is Dr. Robert Ramey's Boost serialization lib, available via `http://www.boost.org`. His library and s11n share a lot of ideas, rather coincidentally. Unlike s11n, however, Robert's library benefits from the massive peer-review effort which goes on at Boost :/.

[15] Reminder: `std::queue`, `deque` and `stack` are not strictly containers - the are *container adapters*. The unusual traversal requirements of queues and stacks make them difficult to serialize efficiently.

- 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 arbitrary types, 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.

- Advanced techniques allow client code to completely reimplement/replace most of the library's underlying layers with their own - without touching the s11n code. For example, class factories or even the client-to-core API translation layer can be replaced by providing certain class template specializations.

- 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 persistence 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 "structure-and-properties"-based system ensures that legacy data do not become invalid until developers[16] *want* them to become so.

- It sports *compile-time type-safe classloading* without the use of a *single* type-cast (neither in the client nor in the library). The classloader is factory-based, and can load just about any classes, including 3rd-party classes, without them knowing they are participating.

- The API is *100% data-format agnostic* and places no file naming conventions client data files. Several (err... many) different data format handlers currently exist, and adding custom Serializers is fairly painless: all you need is an input parser and an output formatter[17]. As of this writing (April 2005) s11n 1.0 has *seven* file-based formats, including three XML dialects, one MySQL-powered "format", and experimental add-on support for ftp/http which can theoretically work with arbitrary file-based formats. That is, as far as i am aware, more formats than any existing serialization library, regardless of implementation language. Why so many? Mainly just to show that it can be done ;).

- Does not impose any special filename conventions or restrictions on clients[18]. That is, if you want to call your saved data `MyData.doc`, go right ahead.

- *All* clients of s11nlite may share serialized data between themselves, regardless of their underlying client serialization interfaces. If they APIs can see each others' factories then they can also transparently fully deserialize each others' data.

- Optional client-transparent zlib and bz2lib file de/compression, for 60-95% file size reduction. When enabled, de/compression happens transparently - usage of s11n does not change one iota.

- 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, but must then supply their own Serializers (i/o handlers, which need not use streams, but could use a relational database or any other back-end).

- 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. This also allows complete separation between the processes of the state storing/restoration and any resulting i/o (the core doesn't even know what i/o *is*).

- Uses only ISO-standard C++ constructs, no compiler-specific extensions.

- Allows clients *complete* control over *how* an object is serialized: s11n makes no assumptions about what you want, it only tries (very hard) to help you meet your data persistance needs. That said, s11n can be told how to serialize many complex object types with very little instruction, so clients need not normally do very much work.

---

[16] Or, admitedly, the all-powerful Marketing Director.

[17] A new Serializer can be implemented in under an hour if one has related Serializer or parser code to start from, and can normaly be done in as little as a few hours even when writing from scratch. The real effort is normally in writing the input parser: the only special consideration normally needed for output is the escaping of, e.g. strings (this is format-dependent).

[18] It might be limited by your underlying filesystem or STL, e.g. in regards to Unicode. s11n has no special support for Unicode, relying on `std::string` for all string operations.

- It comes with an *absurd* amount of documentation, in the form of this document, the API docs and the web site.

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

2.4 WTF is s11nlite?

(WTF is a technical term used very often by I.T. personnel of all types. It is short for "*What the foo?!?*")

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 and, as it is 100% compatible with the core, it 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 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, and we'll help you out.)

Historically, the s11n architecture has been significantly refactored three times, and it has evolved 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 than earlier architectures.

Users new to s11n[19] 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* (but a functionally complete one) of an even more powerful, more abstracted layer known as "the s11n core" or "core s11n."

2.4.1 Repeated warning: *learn s11nlite first!*

We'll say this again because people don't seem to want to believe it...

i wrote s11nlite because i, the author of s11n, found s11n's core "too detailed" for client-side use. i like the general core model, but it is cumbersome to use directly, due to the many places where template parameter types must be specified. So i got tired of dealing with it and sought out a Simpler Way of Doing Things. That is what s11nlite is all about.

If you think i'm kidding about learning s11nlite first, take a look at this note from s11n user Paul Balomiri[20]:

> "I didn't trust you on the point about understanding s11lite first (don't ask why, it was a mistake anyway)."

That is, for the *vast majority of cases*, s11nlite provides everything clients need as far as using s11n goes, and has a *notably* simpler interface than the core library. s11nlite, combined with the various generic serialization algorithms shipped with s11n (e.g. in `listish.hpp` and `mapish.hpp`), provide a complete interface into the framework.

Another point to consider: in client-side code i (s11n's author) use *only* s11nlite and the generic algos/proxies, and almost never dip down into the core, nor do i deal with the Serializer interface from client code. Thus, i can assure you - a potential s11n client - that s11nlite can do almost anything you'd want to do with this library, and is *significantly easier* to work with than the core interface is.

If you still don't believe me, please re-read this section until you do.

2.5 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. Below is a list of 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. Much more detailed information and speculation about the overall client-side costs of deploying s11n-based code can be found in section 20.

[19] At this point, that's basically everyone except me. ;)

[20] As of this writing, Paul uses s11n 1.0.x for some massive data sets: 10 million data points describing the whole street network of Vienna, Austria. :)

- 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. (One user reports that compile times increase by as much as 14 *times* when building with libs11n 0.8.x, but this has been cut drastically since his report.) Some code is in implementation files, so clients must still link to the s11n library, just as they would for any typical C/C++ library.

- Due largely to the side-effects of heavy reliance on class templates, s11n is unsuitable for systems with very limited filesystem space or main memory (e.g. embedded systems, handheld computers, etc.).

- 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 hidden behind very abstract "conventions" and "close approximations". Using the s11nlite interface will completely eliminate most potential "startup problems" when getting used to this library. What is s11nlite? See section 2.4.

- The supplied build tree will only run on GNU-based systems. That is, systems running all the common GNU tools like `make`, GNU `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 - 3.4.x, 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). It is hoped that we will directly support a Windows build tree by the time version 1.2 comes out, as more and more Windows users are showing an interest in the library. Reminder: i don't have Windows, so i can't directly port it. If you give me legal copies of the tools, i will personally do the port.

- s11n is untested with serializing binary data. It "should be possible", but implementing it in terms of the current Serializers (e.g. as string-encoding conversions like base64) would be rather inefficient, i think (even moreso than s11n's normal techniques, i mean). That said, any data which can ultimately be represented as a one or more `std::string` objects and can be structured in a DOM-like fashion (even if only via transformation) should pose no problems at all for s11n. (We avoid binary formats so that we can evade the problems related to machine endianness.)

- The library currently has no algorithms for saving *graphs* - that is, *structures with joins*. This *can* and *has* been done in s11n, but no generic algorithms are (yet) provided for doing so. For more information see section 19.3.

- s11n is untested in multi-threaded environments. See section 19.4 for more details and speculation.

- The library does not publish any strong exception guarantees at the moment. These details are under consideration, and are something which need a lot of serious consideration before deploying. At this point, there is no guaranty that s11n will ever have a specific exception interface. We're open to suggestions from users with strong opinions on this topic (for *and* against it)! :)

- It is driven with Generic Programming and reusability/maintability 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. Try it out and make your own judgement.) Its general model inherently at-least-linear (or even worse), as discussed in more detail in section 20.5.

- s11n's development is primarily steered by my hobbies and my client-side needs, and is constantly under experimentation.

- When statically linking against libs11n, dynamic loading of DLLs will not work. This has something to do with [lt]dlopen(), but i am not certain what. Thus the build process for libs11n builds no static libraries. [TODO: this might not be the case any more, especially since we removed the direct dependency on dlopen(). Something to try out.]

2.6 Version Compatibility

As of the release of 1.0.0, libs11n will attempt to follow the version compatibility guidelines laid out below.

- Major version number: the X in $X.Y.Z$. With Major version increments there are no set guidelines as to what might change, and there are absolutely no guarantees of compatibility with older releases.
- Minor version number: the Y in $X.Y.Z$. Minor number increments may or may not be API-compatible with previous releases. They may change existing conventions or introduce new ones. As per "the Linux convention", odd-numbered Minor numbers represent "development trees", intended for developers and early-adopters. Likewise, even-numbered Minor numbers represent "stable" trees, suitable for client use. Within development trees things may change at any given time - do not rely on them for production work. (That said, for my own client code i typically use the development branch.)
- Patch level: the Z in $X.Y.Z$. Patch-level changes should be binary-compatible with earlier releases in the same Minor number. Binary compatibility will be sacrificed in the interest of "important" fixes or additions, but this should be the exception, not the rule. Within the same *even* Minor number, well-established conventions will never be drastically altered from one patch level to the next (in development trees, anything goes).

s11n's basic model ensures that *data formats are almost always compatible across differing s11n versions*, and that when they are not there is a specific reason for it (it doesn't happen by accident). It is very rare that a *format* ever changes after it's initial definition, and thus data saved with s11n are "almost guaranteed" to be compatible across s11n versions, assuming a given format is not abandon at some point. In cases where such compatibility is broken, i will do my best to release a tool to convert older data files to newer formats. Historically speaking, only once has an s11n-supported format ever changed significantly after its initial release (and two of them have stayed the same since the year 2000). See section 13.2 for more information on the available Serializers.

2.7 Optional supplemental libraries

s11n can make use of the following additional libraries, but does not strictly require them:

- `zfstream`, a published-by-s11n.net lib, provides transparent de/compression for zlib and bz2 data. This library comes as part of the source bundle but is not required by s11n 1.1 and higher (it is required in 1.0.x). Direct dependencies on this library are not recommended, as this library will be replaced once i get my hands on some more flexible code being written by my friend Marc Duerner. If you want zlib/bz2lib compression *now*, however, this is the way to plug it in to s11n.
- `libexpat`, required only if you want to build and use the expat-based XML Serializer (section 13.2.2). This library is almost certainly installed on almost all Unix-like OSes, because it is the *de facto* standard amongst the various Open Source, C-based XML libraries.

3 Main differences between 1.0.x and 1.1/1.2

This section will only be of interest to users of s11n 1.0.x, and summarizes the significant changes from that version (i.e., those which would directly affect users of 1.0). This entire section assumes prior knowledge of how s11n works. If you have never used 1.0, and are just starting out with s11n, skip this section entirely - it is likely of no value to you unless you're a fan of arcane software history.

Version 1.1 is the "development/experimental" branch of libs11n, and what will eventualy become the 1.2 "stable" branch. 1.0.x will continue to be actively supported, and possibly extended in minor ways which do not affect the underlying architecture, for the forseeable future (at least through the end of 2005, probably).

While this section might look quite large, architicturally *very little* has changed since 1.0. However, there have been a number of code reorgs and a few relatively low-impact additions. It is believed that porting from 1.0 will require relatively little client-side work (but some will be required, mainly due to header changes).

3.1 s11n mantra change

Since the beginning, s11n's core mantra has been that s11n is here to *Save Your Data, man!* As is turns out, that is a misrepresentation. Actually... it's a bald-faced lie. The honest truth is that s11n is here to...

> Save *Our* Data, man!

Note the one-letter change, which is more significant than the single missing letter might imply.

3.2 Code consolidation and removal

One of the major goals of 1.1 is to have a tree which will compile on (Microsoft(tm) Windows(tm))(tm) platforms. Another is simplifying support for arbitrary build processes. Yet another related goal is to make the core library more easily forkable, so as to be able to copy it into arbitrary trees.

One requirement for achieving these is some major code refactoring, mainly elimination of all of the "extra bloat" which comes along with the support libs which 1.0 relies upon (that is no trivial amount, due to my packrat-like nature when it comes to utility code).

So, with our sights on portability, and also in the interest of a cleaner build process, the vast majority of the "support libs" have been factored either out or in. That is to say: some of the code (not much) got moved (back) in to s11n and the rest (the majority) was sent packing to CVS limbo. In any case, the s11n core tree is now 100% standalone, with some notes:

- The `zfstream` support lib is used by s11n if it is found, but it is not required. This is the *only one* of the 1.0 support libs which 1.1 now looks for - it no longer uses any of the others which 1.0 relies upon.
- DLL support has been removed from the s11n core, mainly because it poses a slight portability problem at the moment. This will likely be factored back in, in some form, long before 1.2 is released. In any case this will not affect the client-side API, and will affect only two or three small places internally.
- All Serializers which ship with the library will be linked in with the main library, instead of as DLLs. This is primarily in the interest of easing portability to other platforms, but is also currently necessary because the DLL side of the classloader layer was factored out (see above). Note that this does *not* change how the Serializers are *used* in client code, but affects how they are linked in with the main lib: they are still loaded via the dynamic-style interfaces (e.g. `s11nlite::create_serializer("MySerializer")`). If you are the only other person on the planet who actually does dynamically load Serializer DLLs this change will affect you, but if you're doing that then you know what needs to be done to fix it.

3.3 Factory code reimplemented

While the older factory/classloading code (named cllite) is functionally okay, and provides an adequate interface, its code base contains a lot of "evolution cruft". In December, 2004, i was offered a spot on the `pclasses.com` team, to assist them in their 2.x rewrite. The first assignment was to implement a new factory, which i did by taking the learnings from their 1.x factory, s11n's cllite, and some other experimental code. After it proved its worth in the P::Classes tree, i ported a copy into s11n. The newer factory is not markedly improved, functionally, but provides a more focused factory interface than cllite and has a couple new tricks to try out.

(It may be ineresting to know that P 2.x has it's own integrated copy of libs11n. *That's* why i want the s11n code to be easily forkable!)

3.4 `node_traits<>` changes, `s11n::data_node` replaced with `s11n::s11n_node`

To make a long story very short: the `data_node` type was "the original" abstract s11n container[21], introduced in s11n 0.7.0. When the type traits system came along (version 0.9.3), i refactored `data_node` into a slightly more focused API, `s11n_node`. That class has been around since the summer of 2004, but hasn't been actively used within the s11n tree (only for testing the node_traits-related features). As of 1.1.0, `data_node` has been completely removed and replaced with `s11n_node`. Also, `s11n_node`'s API has changed slightly, to make it a bit leaner. Sorry for not having a deprecation period, but making the switch is actually much less painful than it sounds - even trivial (or a no-op) for most client-side code.

What this means for client code:

- Users of the `s11nlite::node_type` typedef normally simply need a recompile (which they would need anyway, because 1.1 is not binary-compatible with 1.0).
- Users of `node_traits<>` iterator-related typedefs and functions will need some slight modifications: don't use the (missing) typedefs, but go through the appropriate sub-typedef, so to say. For example: `node_traits<>::begin()/end()` and `node_traits<>::[const_]iterator` are now `node_traits<>::property_map_` members (they always were, but the "convenience" interface was removed because it was confusing to remember if it refered to the properties or the children).

[21]Not to misrepresent: i mean "the original" as in "the first one to exist in libs11n." The basic model for such containers had been demonstrated as early as summer 2000 in Rusty Ballinger's libFunUtil, if not also in other places, and was used, but in a much different way, in s11n 0.6.x and earlier.

- Clients who explicitely used `data_node` should globally replace that with `s11n_node`. This transition will normally be seemless *if you use `node_traits<NodeType>`* to manipulate your nodes (as is *The One and True Way* in s11n), otherwise other changes *might* be required to accomodate the API differences between the two node types. The APIs are functionally identical, but are *intentionally different* so as to trigger errors in the s11n core code if *it* does not hold to "the `node_traits<>` rule." (That is, the two node types have different APIs to *force me* to fix any s11n core code which isn't using `node_traits<>`!)
- The `data_node.hpp` header of course no longer exists.
- These changes should not affect data files at all, because the two node types are fundamentally the same (only one string identifier in their output is different, but it's not significant for client purposes).

Users who follow the documentation and use `node_traits<NodeType>` to query and manipulate their data nodes, and clients who use template-defined Node Types rather than hard-coded ones, are mostly not affected by this change but may need to make some header-related fixes and a couple typename fixes. e.g. see the notes about about some typedef-related changes and the removal of the `begin()` and `end()` members of `node_traits<>`. Their existence was logically ambiguous, with children and properties both competing for iterator types, and was confusing to remember which iterator `begin()` really returned. `node_traits<>` still contains all of the typedefs and accessors needed to get at that data, but the user will have to go one typedef or function call deeper to get it (but the client code's intention will also be clear to humans, which was not the case before without an additional lookup in the API docs).

3.5 New header conventions, faster compile times

Largely in the interest of bringing some sanity to the s11n build tree, and partly because i have an insatiable urge to hack build processes[22], we have undergone some significant build tree and header reorgs. Again. Yes, i know that's twice... er... three times in the past 12-month period. Learn to think of it "improvement via natural selection" and it doesn't hurt quite so badly. If it makes you feel any better (it does me), the very basic tests i have run show a cut in compile time by as much as 500%. That is, as much as 5 times faster compared to equivalent 1.0 code. Most client-side code will probably see compile times cut by 50%-300%, at least as far as the s11n-side of the compiles goes.

First off, the main Serializable registration header has been renamed: `reg_serializable_traits.hpp` is now called `reg_s11n_traits.hpp`, because that's what the file does - registers `s11n_traits<>`-related code.

Secondly, many headers have been renamed or consolidated into other headers (this mainly affects the i/o and proxy code, but also some of the core algorithms and functors).

The most notable reorg is how the serialization proxies for PODs and STL containers are registered. In 1.0 they were registered *en masse* via headers which included support for multiple containers. This is all fine and good, from an ease-of-use standpoint, but causes measurable (and human-noticable) increases in client-side compile times even for cases where most of the proxies aren't used. In an attempt to decrease client-side compile times, each proxy type now has it's own header. All such headers follow common naming conventions and live in a new header subdirectory:

```
#include <s11n.net/s11n/proxy/std_vector.hpp> // register std::vector<T> proxy
#include <s11n.net/s11n/proxy/pod_int.hpp> // register proxy to promote 'int' type to
a first-class Serializable
#include <s11n.net/s11n/proxy/listish.hpp> // algos and base proxies for list-like types,
but no proxy registration
#include <s11n.net/s11n/proxy/mapish.hpp> // algos and base proxies for map-like types,
but no proxy registration
... and so on...
```

The end effect is that clients must individually choose which proxies they will need. This is slightly unfortunate, but is a one-time cost of including the proper header(s). The main benefit is, for the vast majority of client-side cases, improved compile times. Even in the *worst* cases, compile times should be faster than with 1.0.x because 1.0 tries to install a lot of proxies which are almost never used. If this change *really* annoys users, they may make their own "mass-include" files and include all the proxies they want to. In fact, if compile times are not a concern to you, either because you are extremely patient or because you have access to the lab's Monster Computer, i recommend the mass-include approach, *but only for the sake of ease-of-use* when it comes to figuring out what

[22]Shameless plug: `http://toc.sourceforge.net`

proxies you need. For standard PC users, i don't recommend the mass-include approach at all, at least not unless you are unusually patient while waiting for your code to compile.

i have *attempted* to structure the proxy headers in a maintainable and extendable manner, such that it shouldn't take too much effort to locate the proper proxy header one needs, nor to add new proxies by following the current conventions. If *you* have suggestions for a better layout, please feel free to get in touch! (But be aware that you suggestion might be used, which might of course mean more code reorgs. ;)

3.6 Fetching class names of Serializables

In one of those, "*You utter moron! You should have done this nine months ago!*" moments, the s11n_traits<SerializableTy
interface has been extended to include one static function:

```
static std::string class_name( const serializable_type * HINT );
```

See the API docs, in traits.hpp, for full details, but briefly: this replaces all of the older class_name<> and classname<>() kludgery which has been around since s11n's earliest days (0.2.x or 0.3.x, i think). The end effect is the same, functionally, but this approach fits in cleanly with the rest of the API, whereas the older approach did not (i never did like the old way, but it was necessary for a long time). This approach also allows users of 3rd-party libraries like Qt to use polymorphism-friendly BaseObjectType::className() [or similar] member functions, whereas the older approach did not directly support that at this level of the s11n architecture.

> Design note: i am not at all happy about *not* providing a default of 0 for the HINT argument. However, given the usage of s11n_traits<>, which is only "extended" via template specializations, i also do not like the idea of relying on all specializations to provide that 0 in their interfaces. Also, in the case that it ever becomes useful to make s11n_traits<> a virtual base class, class_name() might become a virtual function (i repeat: *that is theoretically possible*, not a concrete plan), and default parameter values in virtual functions make me queasy, techno-philosophically speaking.

3.7 Client-extendable s11nlite

One of the more interesting additions to 1.1 is a polymorphic class which provides the same API as s11nlite: client_api<NodeType>. This effectively allows users to have an s11nlite interface for custom Node Types or to add custom stream handlers. Sometime before 1.2, s11nlite will be refactored to be based off of this new class, such that clients will be able to subclass it and provide their own class instance to s11nlite (via a back-door-shared-instance-injection technique). This can be used, e.g. to provide network support on top of s11nlite using tools like the experimental code at http://s11n.net/ps11n/ (that code was the primary inspiration for the new class). For example, network-aware extensions to s11nlite can be plugged in to arbitrary s11nlite clients without their code, or s11nlite, even requiring a recompile. If some other desperate coder out there adds, say, Oracle support, your s11nlite client code will be able to use it without explicitely having to know about it. Consider, too, that we can actually use factories to dynamically load arbitrary instances of the client_api<>. Weird, eh?

3.8 ~/.s11nlite config file removed: default Serializer class no longer persistant

In 1.0, s11nlite saves its configuration when the library shuts down. While this is all fine and good for a system where only one app uses s11nlite, it causes interference when multiple apps share s11n. For example, when App A sets s11nlite::serializer_class("MySerializer"), App B is going to get that default the next time it starts unless it sets its own (which might then affect App C... *ad nauseum*). Thus we take the simple route and remove it. The only affect this has on clients is that they might want or need to set a default Serializers when their app starts up, using s11nlite::serializer_class().

While the majority of s11n users use the library in only one source tree, i currently use it in no less than six projects, and have often experienced problems with each app imposing its own idea of a default file format on the other apps. So, like so many other dead-ends of evolution, ~/.s11nlite is gone.

Since the s11nlite config object was never really advertised as a feature, it is thought (hoped) that this change does not affect most clients.

Note that the serialization of an application-wide config file is trivial, but that techniques like finding a user's home directory are platform-specific (even under Unix, $HOME is not always the user's home).

3.9 Exceptions conventions

As of version 1.1, i've finally started seriously working on defining exception conventions for the framework. i've decided on a mask-based system, such that the framework will throw for specific types of failures if it is told that to throw for that type. This is all experimental and up for any number of changes. See `exception.hpp` and section 4.7 for details.

4 Core concepts

Users who want to *fully* understand s11n should read this section carefully - here we detail the major components of, and terms used within the context of, the s11n architecture. Understanding these is critical if one wants to *truly* understand how the library works. That said, a *lot* can be done client-side without understanding *anywhere near* all of the gory details: one can get quite far by simply copying example code!

4.1 Terms and Definitions

Below is a list of core terms used in this library. The bolded words within the definitions highlight other 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.

Note that some terms here may have other meanings outside the context of this software, and those meanings are omitted for clarity and brevity - here we only concern ourselves with the definitions as they pertain to us as users of s11n.

- **s11n** - several meanings:
  - A short-hand form of the word "serialization", used in many contexts.
  - The literal name of this software.
  - 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::s11n_node` type, and core s11n supports any types which conform to the conventions laid out by that type. (The core doesn't actually know that type exists.)
  Note that using a Data Node's API directly from client code is discouraged. Please prefer the API provided by `s11n::node_traits<DataNodeType>` instead, as described in section 6.1.
- **Node Traits** (`s11n::node_traits<NodeType>`)- an interface for interacting with **Data Nodes**. Conceptually similar to the standard library's `char_traits<char_t>`. See section 6.1.
- **serializable** (with a *small* "s")- the property of being able to be saved and to restore state. For example, to allow persistent object 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 Traits** (`s11n::s11n_traits<SerializableType>`) - a type for encapsulating s11n-related information about a **Serializable Type**. See section 6.2.
- **Serializable Proxy** or **Serialization Proxy** - a functor (optionally 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 the proxied type are delegated to the proxy type. This allows, amongst other things, transparent serialization of 3rd-party classes and drasticaly simplifies the serialization of containers.
  Proxies are *not* **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** . In this library that is accomplished by storing the state in a **Data Node**, which is conceptually identical to storing *a copy of it* in an STL container.
- To save a **Data Node** to a data stream via a **Serializer**. Stream-related serialization is normally called "saving".
- Several other subtle, 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. Stream-related deserialization is normally called "loading".

- **de/serialization** or **de/serialize** - shorthand forms of "deserialization and serialization" and "deserialize and serialize."

- **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 dealing with 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.

- **Serializer** - a type responsible for converting **Data Nodes** to and from a specific grammar (i.e., a data format). For example, some Serializers use an XML dialect while others use custom formats. Theoretically, any data which can be structured in a DOM-like fashion (even if only via logical transformation) can be handled by Serializers. In s11n *Serializers* are also always *Deserializers* (at least logically, in terms of the API interface).

- **serialization operators**, `de/serialize()`, or **Serializable Interface** - generic names for a 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 core library's `de/serialize()` functions.

- **de/serialization operations** - 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. If we needes a really technical definitiion, this would be pretty close to correct: any operations which end up forwarding through the `s11n_api_marshaler<>` (SAM) internal interfaces (section 15).

- **Default Serializable Interface** - Serializables which implement both of their serialization operators as `operator()` , and which follow the conventions laid out in section 5, are said to implement the *Default Serializable Interface*. Types which do this do not need to tell s11n what their serialization interface looks like - we will be able pick them up automatically.

- **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 deserialization). The s11n classloader has support for loading classes from DLLs, but that feature is not covered much in these docs because its operation is transparent to the API. Classloaders work primarily not off of specific "concrete" types, but off of **Interface Types**, as described briefly below. See the docs available via `http://s11n.net/class_loader/` for more detail than you probably want to know about these, or a summary paper at: `http://s11n.net/papers/#classloading_cpp`

- **T's classloader**, or **the T classloader** - Refers the the classloader (factory) which uses type T as it's point of reference for registering and loading classes. More specifically, it (currently) means `s11n::fac::factory_mgr<T>`, though the exact factory implementation which s11n uses is not a defined part of the public interface. For proper polymorphic deserialization, subtypes of T should be registered with T's classloader, regardless of whether or not they also register with their own classloader (e.g. `factory_mgr<SomeTSubType>`).

- **Interface Type** [note: in s11n 1.0 these are referred to as Base Types, which is marginally incorrect and definitely more ambiguous than Interface 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 Interface Types with the Interface Type's **Serializable Interface**, and is critical for classloading purposes. It is covered in massive detiail in the classloader's library manual. In a broader sense, Interface Types are used as contexts for marshaling the s11n and client-side **Serializable Interfaces** into internally-compatible forms. Interface Types are often used as contexts, in the form of template parameters for functions and class templates [23] for which this Interface Type distinction is significant (e.g. those dealing with de/serialization and classloading). The abstract topic of Interface Types is covered in more detail in a paper written as part of this project: `http://s11n.net/papers/#classloading_cpp`

- **Streamable [Types]** - In the context of s11n this means any type for which `ostream<<` and `istream>>` operators can be applied to successfully save and 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[24]. This also implicitely *excludes* all pointer types (but note that s11n often handles objects of type (SerializableT) and (SerializableT *) equally). **Serializables** are *not* implicitely Streamable, as s11n does not deal with streams at it's core, and thus the **Serializable** interface is stream-ignorant.
- **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 technically limited to, forwarding requests to **Serializable Proxies**. SAM allows clients to transparently proxy the s11n interfaces, as covered in section 15. Clients will almost never have to know about SAM, but it does play a significant role internally.
- **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 `s11n::de/serialize()` variants. Everything else, from the **Serializers** to the s11nlite interface, is built around this *tiny* core.

Did you get all that? Don't worry - you don't need to memorize it, but if you find yourself confused by a term in this documentation, try looking it up in the list above.

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.

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 documentation. Briefly, they are:

- **i.e.** - "in other words" or "in effect" (from the Latin *id est*[25])
- **e.g.** - "for example" or "example given" (from the Latin *exempli gratia*)
- **Algorithm** - we use the same general meaning as in common STL usage: a computation, normally one which is genericized in form such that it can be applied to a wide range of types which meet a published set of conventions for that algorithm. Like **functors**, understanding algorithms is essential to effectively using the STL, and the two often go hand-in-hand.
  For *numerous* well-published examples of algorithms see those in the STL itself, defined in the ISO-standard `<algorithm>` header file. s11n includes many serialization-related algorithms and functors.
- **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.
- **ODR, the One Definition Rule** - C/C++'s rule which, put simply, basically states that no type may be *defined* (i.e., implemented) more than one time in any given binary or library. This is not an arbitrary rule, but a technological limitation, akin to `std::map` being able to contain no more than one object with a given lookup key. In any case, it's rather a *sane* behaviour, if you ask me.
  In s11n ODR is an oft-heard term because its template-based nature, in particular its 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's trivial to avoid once you seen it happen it a few times and understand it's nature.) With the release of version 0.8.0, all commonly-occurring ODR-related problems are believed to be solved. (i haven't personally seen an s11n-caused ODR violation since the 0.8.x series, except when i have incorrectly double-registered a proxy.)

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

[24] s11n do not automatically register proxies for these Streamables. For more info see `pods_streamable.hpp` or grep the Changelog/sources/this file for that filename. See also section 12.1.1.

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

- **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[26]. Additionally, the amount of SP a given reward or pentalty gives or takes may be adjusted by the relative experience levels or repution 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[27] they invariably begin, at some indefinate point, to concentrate on Style as much as they do on the nature of 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 generaly "pick up SP" much more readily than greenhorns can.

4.2 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[28]*:*

> 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 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, additionally, 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*[29].

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 1.1, the core's relationship with classloading has been reduced to the factory side. Put simply, this means no automatic DLL support, but the factory model makes adding one's own DLL support quite simple.

- `s11n::s11n_node` - this is the reference implementation for the **Data Node** concept. This is supported by all of supplied node-related algorithms and functors, though they actually have no direct dependencies on it. It is considered poor style use call the Data Node API directly from client code - using the `s11n::node_traits<NodeType>` interface is *highly* preferred, for compatibility with 3rd-party node types and for future compatibility with new node types. For example, s11n 1.0.x uses a different node type (`s11n::data_node`), and using the `node_traits<>` to access nodes makes this transition transparent.

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

[27] As developers, of course, not necessarily as human beings.

[28] 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 a Style Point? See section 4.1.

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

- **Serializers** - these objects are responsible for marshaling Data Nodes to and from specific file formats (grammars). The library currently s11n ships with several Serializers, supporting a variety of data formats. All Serializers shipped with s11n are available to s11nlite, but s11nlite restricts itself, for purposes of *saving* data, to one of them (*which one* it uses is not strictly defined by the interface, and may easily be defined by the user). s11nlite does not need to be told what format to use for loading, as that is determined dynamically (see sections 13.1.1 and 13.1.3).

- **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. It may be interesting to note that the core API is made up of less than 50 lines of code.

- **Serialization API marshaler** (SAM)- the core de/serialize functions pass all of their request through this internal layer. These types can be swapped out transparently, customizing the serialization interface on a per-base-type case. This feature is used, for example, to direct serialization through Serializable Proxies, or to implement pointer-to-reference type translation as needed. These marshalers 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. SAMs can then do almost whatever they like with the API, except change parameter constness for nodes and serializables - they may add arguments as they wish! This can be used, e.g. to implement framework-enforced data versioning
SAM is covered in section 15.

- **Type Traits** (section 6) - as of version 0.9.3 these types are used to encapsulate interface information for Serializables and Data Nodes. Users of the STL may be familiar with standard traits types such as `iterator_traits<>` or `char_traits<>`. The s11n traits types, `s11n_traits<>` and `node_traits<>`, play similar roles as those types do in the STL. Note that `s11n_traits<>` and SAM overlap is some ways, as described in section 15.

- **s11nlite** - a tidy little interface providing a wrapper around the above layers, 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* direct knowledge of libs11n, helping to support the "non-intrusion principal" which s11n works hard to uphold. For an example of this, see the P::SIO module in the P::Classes 2.x source tree (via http://pclasses.com), where we have implemented a custom s11nlite-like interface to suit the needs of that project better.

- 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 at compile-time.

4.3 Process Overview

4.3.1 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. a Data Node) via the s11n serialization interace (e.g. `s11nlite::serialize()`).
2. s11n proxies the request to the registered Serializable Interface and passes the target Data Node and source Serializable to the registered interface.
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 Serializer's 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, where a child Serializable is serialized.

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

4.3.2 Deserialization

A client-initiated deserialization request in s11n normally looks more or less like this:

1. Client requests the deserialization of a Serializable Type from a data stream/file.

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. Obviously, if there is no Node then processing stops here with an error (typically, false or `0` is returned).

4. s11n looks at the root Node to determine which Serializable Type to instantiate. If it fails to find a class, or cannot instantiate the requested type, processing stops with an error (typically, false or `0` is returned).

5. s11n marshals the data-to-be-deserialized to the registered (De)serialization Interface for Serializable's type.

6. Deserialize operator's implementation should restore the Serializable's state from the source Data Node. If it returns false then processing stops.

7. s11n destroys the now-unnecessary Data Node.

8. s11n returns a (Serializable *) to the client, which the client now owns.

The interface also supports deserializing nodes directly into arbitrary Serializables, effectively bypassing the first four of the above steps. Also, clients may stop at point 7, if they are only interested in the data, as opposed to wanting the objects the data represent. For example, the `s11nconvert` and `s11nbrowser` applications (sections 16.1 and 16.2) never rely on a specific Serializable Types.

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

When saving data each node is given a name, fetchable via `node_traits<NodeType>::name()`. 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 having *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 the syntax conventionally used by XML nodes and in most programming languages:

> ***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 most of the parsers supplied with this library (even funxml and simplexml, but not expatxml), but the data won't be portable to more standards-compliant parsers. Thus, if data portability is a concern, avoid numeric keys altogether.

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.

4.5 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.

- Data Nodes are the basic types used to store arbitrary key/value pairs and child objects. They follow a DOM-style interface, so their usage is fairly straightforward. The core library and generic algorithms support any Data Node type which can be proxied via `s11n::node_trails<>` (section 6.1).
- The entire client-side interface for loading and saving all objects is declared in `<s11n.net/s11n/s11nlite.hpp>`, in the `s11nlite` namespace. The core code, and many node-related functors and algorithms are available in these namespaces: `s11n`, `s11n::list`, `s11n::map`, `s11n::va`. That said, clients may directly use the core s11n, bypassing s11nlite completely, but using s11nlite is *highly* 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, we serialize data to and from 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, so clients who don't care about the underlying data format need never worry about this highly overrated detail.
- Classloaders and their "InterfaceType" types are important concepts to understand in s11n, mainly for template-types reasons. They are covered *in detail* in the classloader 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 s11nlite use cases won't require much understanding of the subtleties of the classloader framework.

4.6 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 detailed, but here's a summary:

- SOME error value is needed. Integer values must either be mapped to a known set of error codes 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 *almost never* done, at least 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 cases!

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

- The vast majority of the client-side part of s11n doesn't work 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 yada yada yada).
- The s11n core is container-based, and container insertions and extractions, as a general rule, do not fail. Also, container searches only fail in the sense that the sought-after data simply isn't there.
- In the event of a stream- or grammar-level input failure 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[30].

Furthermore, some specific operations, such as using `std::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 no 100% satisfying solution, and bools seem to meet the middle ground fairly well.

[... much later ...]

As of version 1.1 we've introduced an exception mechanism similar to that used by standard i/ostreams: we allow the client to specify a mask representing the types of failures he would like to have exceptions for. More info about this is in section 4.7.

4.7 Exceptions conventions

As of version 1.1, s11n attempts to define a set of exception-related guarantees, such that we can define the state of, e.g. a container, when the de/serialization of a child node fails. The support is not yet complete, but at least it's finally started.

The base-most exception type for the framework is, naturally enough, `s11n::s11n_exception`, which derives from `std::exception` and follows the same interface.

The API does not have any `throw(xxx)` specifiers on most functions. This is to allow the library to propagate user-thrown exceptions without running the risk of `unexpected()` being called (that's C++'s way of crapping out if a function throws an exception which does not match its `throw(xxx)` specification.)

The API now has a pair of `s11n::throw_policy()` getter/setter functions, which can be used to query and set the current throwing policy. The policy is specified as a bitmask of types of errors which should trigger exceptions. The bitmask is defined in the `s11n::ThrowPolicyMask` enum, but the API actually uses `throw_policy_enum`, which is a typedef for an unsigned integral type, to allow us to "extend" the mask without touching that enum.

[30] Especially when s11n's author cannot 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. :/

The core de/serialize functions honor the current `throw_policy()` and behave accordingly. That is, on failures they return false if the policy suggests not to throw and they throw if the policy says to. Failures which do not trigger exceptions fall back to the normal error handling for the given routine, which normally equates to returning false or null.

Proxies and convenience functions which recursively serialize almost all go through the two core de/serialize functions at some point, and therefor will also throw if the policy says to. Several existing algorithms have been changed to provide no-change-on-throw guarantees, and any algos where this is not the case will be fixed as we come across them.

The exact guarantees regarding any throw behaviour are necessarily documented on a per-algorithm basis, so see the appropriate API docs. As mentioned above, almost all recursive routines go through the core de/serialize and may throw, but the exact definition of what happens in the face of exceptions must be defined by the algorithm.

Note that no amount of conventions will 100% transparently protect clients from problems such as memory leaks. As a simple example, if a deserialization of list<T*> fails halfway through and throws, the list is left half-populated and needs to be cleaned up somewhere (presumably by the client who tries to deserialize it, though it would arguably be useful for the list deser algo to clean up the whole list in a failure case.

5 Serializable Interfaces: overview and conventions

Rather than overload you with the details of this right up front, we're going to *grossly oversimplify* here - to the point where we're almost lying - 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. The s11n interface is designed such that it is very difficult for clients to have an environment where 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 11.

Trivia:

> When the de/serialize operators are implemented in terms of `operator()`, with the above-shown signatures, a type is said to conform to the *Default Serializable Interface.*

5.1 Serialize operator conventions

- If the type is polymorphic, it **must** set it's class name in the node, e.g. using `node_traits<NodeType>::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 class name needs to be called for them (SAM does this - section 15, and proxies often re-set it). If this operator calls a parent type's serialization operator, the class name should be set *after* calling the inherited operator, such that the *subtype's* class name is stored.
- 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.

5.2 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 leaking resources or duplicating container entries.
- Returns true on success, false on error.

5.3 Data Node class names (IMPORTANT!)

Let us repeat that many times:

> *while(! this->gets_the_point())*
>
> > *std::cout << "The importance of class_name() 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.)

`class_name()` is part of the `node_traits` 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 `class_name()` 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 classloader documentation for hints on what aliasing can potentially do for you.

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

- Serializables (or their proxies) must set the target node's `class_name()` 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. When using the default Serializable registration techniques, you should pass the class name defined in the S11N_TYPE_NAME macro passed in to the registration supermacro (section 11.6).
- If a Serializable class inherits serializable behaviour from a parent type, the subclass must set `class_name()` *after* calling the parent implementation, to ensure the proper subclass type gets into the node. Also, if the parent's operation fails, the child should normally immediately return false.

Some algorithms parse data directly from data nodes, irrespective of the node's `class_name()`, and this is perfectly kosher. 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 **??**.

5.3.1 Example of setting a node's class name

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

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

```
s11n::node_traits<DataNodeType>::class_name( node, "A" );
```

In B's we should do:

```
if( ! this->A::operator()( node ) )[31] return false;
s11n::node_traits<DataNodeType>::class_name( node, "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[32]. To extend the above example, after the node contains B's state, we can do this:

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

(Note that we call `deserialize<A>()` with `A` because that's the Interface Type which registered with s11n.)

That creates a (*B**) and deserializes it using B's interface. Why? Because `node`'s `class_name()` is "B", and the A classloader will load a B object when asked to (assuming it can find B - if it cannot it will return zero/null).

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 because there is no implicit conversion possible from A to B. It will fail at *compile time*. 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 Interface Type for classloading/deserialization purposes, 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*).

5.3.2 Using local library support for `class_name()`

Some heavily object-oriented libraries, like Qt (www.trolltech.com), support a polymorphic `className()` function, or similar, to fetch the proper, polymorphic class name of an object. If your trees support this, *take advantage of it*: set the node's class name one time in the Interface 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 100% reliable way to catch them all has yet been discovered. (*Hear my cries, oh mighty C++ Standardization Commitee!*)

5.4 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 functions directly! Instead they should use the various `de/serialize()` functions. This is to ensure that interface translation can be done by 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 three* 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 serialize and deserialize implementations. In those two cases it's perfectly acceptable to do so, and in fact could not be done any other way. The final case is when you want or need to bypass the internal API marshalling. Any other usage 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 probably can*[33]).

For example, instead of using this:

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

[31] See section 5.4 for why you should never directly call a Serializable's serialization API. This particular case is one of two which simply cannot be avoided.

[32] That is, assuming the subtypes are properly registered with the classloader.

[33] Alas, unless, you have some unusual needs, e.g. you need customized recursive de/serialization to go around the internal marshaling process.

use one of these:

```
s11nlite::serialize( my_data_node, myserializable ); // YES! Friendly and 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 particual case C++'s normal automatic argument-to-template type resolution suffices to select the proper types, so specifying them via `<>` syntax is unnecessary.

Aside from the above cases, the only other "acceptable" case for calling a local de/serialization API directly is when you need to go around s11n's internal marshaling.

In terms of Style Points (section 4.1), calling a Serializable's API directly, except where specifically necessary, is immediately worth a good -1 SP or more, and may forever blemish one's reputation as a generic coder. To be perfectly clean, though, calling the local APIs directly *does not* have any direct effect on s11n. This convention is primarily to help ensure portability of serialization functionality between disparate s11n-powered objects.

5.5 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 );
```

and they use the `s11n::node_traits<NodeType>` interface to query and manipulate the nodes, then their Serialize methods 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 including (no pun intended) the implementation code in the header file.
- Compilers do not completely check template functions until they are called, so there might be a compile-error-in-waiting as coders tweak bits without testing them (what, me? ;).
- Member template functions 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 known effective way to proxy requests for class templates, e.g. STL containers, as it allows a single pair of functions to handle de/serialization for a whole family of types. e.g. two functions which can handle `list<int>`, `list<double>`, `list<char>` ...
- Common C++ literature suggests that smart compilers can eliminate at least some of the middle-man code in many common functor-related constructs.

6 Type Traits

In version 0.9.3 a Type Traits-based system was added to the framework to encapsulate information about Data Node and Serializable interfaces.

The traits types live in the namespace `s11n` and are declared in the file `traits.hpp`.

In short, the traits types encapsulate information about Data Node and Serializable types. Anyone familiar with the STL's `char_traits<>` type will find the s11n-related traits types similar.

6.1 `s11n::node_traits<NodeType>`

`node_traits` encapsulates the API of a given Data Node type. Using this approach it is possible to add new Data Node types to the framework without requiring clients to directly know about their concrete types.

The complete API is documented in the `node_traits` API documentation.

Note that it is considered "poor form" to directly use the API of a given Node type in client code - use the traits type when possible.

The default `node_traits` implementation works with `s11n::s11n_node`. Using `node_traits` to manipulate these objects will ensure that client code can be used with either one (plus potential future node types).

It might be interesting to note that s11n has been used successfully with at least 3 data node types, so the swapping-out-node-type idea has shown to be more than a theoretical feature.

6.2 `s11n::s11n_traits<SerializableType>`

`s11n_traits` encapsulates the following information about a Serializable Type:

- Serialization Functor (typedef `serialize_functor`) - a functor type responsible for handling calls to `serialize()` on behalf of `SerializableType`.
- Deserialization Functor (typedef `deserialize_functor`) - a functor type responsible for handling calls to `deserialize()` on behalf of `SerializableType`. This is normally the same type as the Serialization Functor, but sometimes it is necessary or desirable to implement different functors for each operation.
- Factory Type (typedef `factory_type`) - a functor which is responsible for creating new instances of the type (polymorphically, if required). This allows clients to easily install their own factories for a given class hierarchy, as opposed to being forced to use the default ones used by s11n.
- A single static function, `class_name(const serializable_type * HINT)`, added in version 1.1, allows algorithms to query Serializables for their class names in a more coherent way than in previous s11n versions (but with essentially the same effect and limitations *vis-a-vis* polymorphism).

The interface and its conventions are documented fully in the `s11n_traits` API documentation.

Note that this type has no data members. That said, a specific traits specialization is free to expand the type. For example, it may contain the implementation for the de/serialization operators and typedef *itself* as the `de/serialize_functor` types (yes, this has been done before).

The original intention of `s11n_traits` was to replace SAM (section 15). As it turns out, SAM's (T*)-to-(T&) translation fairly tricky to introduce via traits without an undue amount of extra code (potentially client-side). Since SAM does this in only a few lines of code, as is zero-maintenance (since almost 1 year), the pointer/reference translation support will stay in SAM. SAM is, however, implemented in terms of `s11n_traits`. That actually ends up giving us another layer we can hook in to, anyway, which gives us a bit more flexibility in swapping out components.

7 How to turn JoeAverageClass into a Serializable...

Before we start: the s11n web site has a number of examples of Serializable implementations. You may want to check there if this section does not help you.

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

1. Create the class, implementing a pair of de/serialize methods with the signatures expected by s11n. The de/serialize operators may be defined in a separate (proxy) class in many common cases.
2. Tell s11n that your class exists, via registering it - see section 11.

If you are proxying a well-understood data structure for which a functor already exists to de/serialize it, step one disappears! An example would be proxying a `std::list<int>` or `std::list<Serializable*>` - those are both handleable by the `s11n::list_serializable_proxy` class, provided that the contained types are Serializables. For a list of some useful proxy functors see section 12.

7.1 Create a Serializable class

The interface is made up two de/serialize operators. Types with different interfaces can also be used - see the next section. This library does not impose any inheritence requirements nor function naming conventions, but for this simple example we will take the approach of a serializable object hierarchy.

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 {
    // serialize:
    virtual bool operator()( s11nlite::node_type & dest ) const;
    // deserialize:
    virtual bool operator()( const s11nlite::node_type & src );
    // ...  our functions, etc.
};
class MySubType :  public MyType {
    // serialize:
    virtual bool operator()( s11nlite::node_type & dest ) const;
    // deserialize:
    virtual bool operator()( const s11nlite::node_type & src );
    // ...  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 cannot 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: see section 11 for how tell s11n about the class.

7.2 Specifying custom Serializable interfaces for InterfaceTypes

If MyType does not support the default interface, but has, for example:

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

The library can still work with this. How to register the type as Serializable is described in section 11.

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.

7.3 Specifying Serializer Proxy functors

This is one of s11n's most powerful features. With this, any type can be made serializable *without editing the class*, 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()( DataNodeType & dest, const SerializableType & src ) const;
```

Deserialize:

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

Notes about the operators:

- Yes, both functions "should probably" be `const` in this case, for the widest functor reusability, but if C++ will let you get away with non-`const` operators in your contexts then s11n will accept them.
- The operators may be templates and/or the functor may be a template. As long as C++'s 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. (Trivia: in practice this has only once been necessary, and was probably caused by my mis-use of a non-`const` object.)

We must then register the proxy, as explained in section 11.6.
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 and a proxy as being the handler for a given type, nor may you register two proxies as being the proxy for a single Interface 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, with the exception of those with unusual traversal rules, like `queue` and `stack` (those two could be done, but would require an extra copy to be made, since we may not modify the source object during serialization).
- 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[34]. InterfaceType, 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*.
- In theory it may sometimes be necessary, due to `const`-vs-non-`const` ambiguity, to split a de/serialization functor into two functors. In practice it's happened once, ever, back in s11n 0.7.x.
- 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 introduces or verifies a version property and then passes on the call to the local Serializable interface of the object.
- Client code can, e.g. use a macro to define which proxy will be used for a given type (or group of types), allowing them to switch freely between serialization implementations on a per-type basis. This is how all of the "standard" proxies are implemented.

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

8 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:

- It implements a serializable interface we can register with s11n.

or:

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

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

It is best shown with an example, where we proxy a client-supplied type:

```
#define S11N_TYPE MyType
#define S11N_TYPE_NAME "MyType"
// [de]serialization functor, only for proxied types:
#define S11N_SERIALIZE_FUNCTOR MyTypeSerializationProxy
// optional DEserialization functor, defaults to S11N_SERIALIZE_FUNCTOR:
// #define S11N_DESERIALIZE_FUNCTOR MyTypeDeserializationProxy
#include <s11n.net/s11n/reg_s11n_traits.hpp>
```

You're done!

That's all that's necessary to take complete control over the internals of how s11n proxies a class.

This process must be repeated for each new type. The `S11N_xxx` macros are all unset after the registration header is included, so they may be immediately re-defined again in client code without having to undefine them first. Other proxy registration supermacros may implement whatever interface they like, with their own macro interfaces, allowing per-proxy-per-Serializable customization via macro toggles.

The registration process, on the surface, looks... well, *awkward*. Trust me, though: the benefits over of this simple approach macro- and code-generation-based solutions are tremendous, and have helped make some extremely tricky (or essentially impossible) cases much simpler to implement.

Note that when registering template types, you also need to register their *templatized* 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[35]. However, `list<int>` and (`list<int>*`) are equivalent for most of s11n's purposes.

8.1 `JoeAverageClass<>` class template

The process for plugging in a Serializable class template is demonstrated on the s11n web site:

`http://s11n.net/s11n/sample3.php`

The process is fundamentally the same as for non-templates, with of course... the additional complication of template parameters to deal with. Hacker Intuition aside, the process is quite straightforward and easy to implement, and demonstrated on the above-listed web page.

Optionally, take a look at the standard proxies for the STL list/map containers, in the s11n source tree under `src/proxy/reg_{list,map}_specializations.hpp`. These files demonstrate the serialization proxying of class templates.

9 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 [or you upgrade to a newer s11n which breaks old APIs or conventions].

Remember that when using Data Nodes, it is *strongly preferred* to use the `node_traits<NodeType>` interface, as opposed to the Node Type API directly, as explained in section 6.1. Client code may of course use typedefs to simplify usage of `node_traits`.

In the examples shown here we will assume the following typedef is in effect:

```
typedef s11n::node_traits<NodeType> NTR;
```

[35]That said: STL containers of PODs are handled without any special client-side registration.

9.1 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:

```
NTR::set( node, "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 cases involving ambiguity between ints/bools/chars which may require that the client explcitely specify the property's type as a template parameter:

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

Each property within a node is unique: setting a property will overwrite any other property with the same name.
It must be re-iterated that `set()` **only** works when setting values which are *Streamable Types*. That is, types which support two complementary `ostream<<` and `istream>>` operators. To save Serializable children use the `serialize()` family of functions.

9.2 Getting property values

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

```
T val = NTR::get( node, "property_name", some_T_object );
```

e.g.

```
this->name( NTR::get( node, "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 `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 (or known-bad!) value to use in case the supplied object 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 the `set()` documentat, above, for the proper workaround.
As with `set()`, `get()` **only** works with *Streamable Types*. To restore Serializable children, use the `deserialize()` family of functions.

9.2.1 Simple property error checking

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

```
int foo = NTR::get( node, "meaning_of_life", -1 );
if( -1 == foo ) { ... error: we all know it's really 42 ... }
std::string bar = NTR::get( node, "name", std::string() );
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 invalid or error values.
Theoretically, installing a Serializable Proxy for a type which does such checks and then passes the call on to the object's local Serializable Interface is one way to keep this type of code out of the classes.

9.2.2 Saving custom Streamable Types

This is a no-brainer. Streamable Types are supported using the same get/set interface as all other "simple" properties. Assume we have a Geometry type which support i/ostream operators. In order to save it we must simply call:

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

and to load it:

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

or maybe:

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

9.3 Finding or adding child nodes to a node

Use the `s11n::find_child_by_name()` and `s11n::find_children_by_name()` functions to search for child nodes within a given node. Alternately, use `node_traits<NodeType>::children()` function to get the list of it's children, and search for them using criteria of your choice.

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

9.4 Serializing Streamable Containers

Streamable Containers are, in this context, containers for which all stored types are *Streamable Types* (see 4.1). 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( node, "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 *not* 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( node, "subnode_name", my_map );
```

(Can you guess which two characters changed? ;)

If you want to de/serialize a `std::list` or `std::vector` of Streamable Types, 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). We actually rely on C++'s strong typing to do the hardest parts of type determination, and we don't actually need the type name in some cases involving monomorphic Serializables. More specifically, whenever no classloading operation is required, the class name *ist uns egal*[36].

Note that these functions only work when the contained types are Streamables. If they are not, use the `s11n::list::serialize_list()` and `s11n::map::serialize_map()` family of functions. Note that those functions also work for Streamable types as long as a proxy has been installed for those Streamables.

[36]German for "frankly, darling, we don't give a damn."

9.4.1 Trick: "casting" list or map types

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:

```
data_node n;
s11n::map::serialize_streamable_map( n, imap );
s11n::map::deserialize_streamable_map( n, dmap );
```

Or, more simply:

```
s11nlite::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. The compiler will almost certainly warn you in such a case.

Similar conversions will work, for example, for converting a `std::list` to a `std::vector`. For example:

```
list<int> ilist;
vector<int *> ivec;
// ... populate ilist ...
s11nlite::s11n_cast( ilist, ivec );
```

That's all there is to it. The library takes care of allocating the (int*) children of the vector. The client is responsible for deallocating them, just as one would when using any "normal" STL container of pointers. One simple way to deallocate them is using `free_list_entries()`:

```
s11n::free_list_entries( ivec );
```

Curiously enough, that function will also work on a list containing non-pointers, clearing out the list. This is not an *effecient* way to clear a non-pointer-holding lists, due to a tiny internal detail, but the function supports both types of lists to help support generic algorithms which work arbitrary lists. Ditto for `free_map_entries()`.

9.5 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.

9.5.1 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 `s11nlite::node_type` object:

```
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 (tips: see `s11nlite::serializer_class()` for a way to override which Serializer it uses).

Loading an object is fairly straightforward. The simplest way is:

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

InterfaceType must be a type registered with the appropriate classloader (i.e., the InterfaceType classloader) and must of course be a Serializable type. To illustrate that first point more clearly, ***the following are not correct***:

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

Will not compile: there is no implicit conversion from `InterfaceType` to a subtype of that type.

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

Will compile but will not do what is expected, because it's trying to use a different classloader and API marshaller than `InterfaceType`.

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 any deserialize-related 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 operation fails then `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.

9.5.2 Containers of Serializables

This subsection exists only to avoid someone asking, "how do I serialize a list<T> or list<T*>?"

Here you go:

```
#include <s11n.net/s11n/proxy/listish.hpp> // list-related algos
#include <s11n.net/s11n/proxy/std_list.hpp> // std::list<T> proxy registration
...
s11n::serialize( target_node, src_list );
...
s11n::deserialize( src_node, tgt_list );
// or:
ListType * tgt_list = s11n::deserialize<ListType>( src_node );
```

The same goes for maps, except that you should include `mapish.hpp` and `std_map.hpp`. Note that "list" algorithms actually work with `std::list`, `vector`, `set` and `multiset`, but that proxies for each general list type must be installed separately, by including one of `std_{list,set,vector,...}.hpp`. The map algorithms work for `std::map` and `multimap` and are proxied via the headers `std_{multimap,map}.hpp`.

So what is different from the above code and de/serialization of any other Serializable type? Nothing. That's part of what makes s11n so easy to use - clients only really need to remember a small handful of functions.

9.5.3 "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 (i.e., it's `class_name()`) 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" ) ) {
    if( ! s11nlite::deserialize( *ch, myobject ) ) {
        ... error ...
    }
}
```

The notable down-side of brute-force deserialization, however, is this: if the deserialize operation fails then `myobject` may be in an undefined state. Handling of this is (a) very client-specific, and (b) in practice it is very rare for a deserialization to fail at this level. Brute force deserialization specifically opens up the possibility of feeding any data to any deserialization algorithm, which of course means that for correct results you must use matching data and algorithms.

10 Walk-throughs: imlementing Serializable classes

TODO: REWRITE FOR 1.1: headers are 1.0-style here

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 under `src/client/...`.

The s11n web site has several examples, going well beyond what is presented here.

10.1 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.

10.1.1 The #includes

We will need to include the following headers for our particular case:

```
#include <s11n.net/s11n/s11n.hpp> // core framework.  Or use s11nlite.hpp.
#include <s11n.net/s11n/list.hpp> // list algos/proxies
#include <s11n.net/s11n/map.hpp> // map algos/proxies
#include <s11n.net/s11n/pods_streamable.hpp> // installs proxies for basic PODs
#include <s11n.net/acme/algo.hpp> // utility functions used during deserialize
```

The `pods_streamable.hpp` header was added in 0.9.14. As of that version, PODs are not registered by default, because doing so eats up huge amounts of compile time and they are not used from most client code. Clients wishing to promote most basic PODs to "Full Serializable" status should include that header or implement similar registration code of their own.

10.1.2 The data

Let's assume that MyType has 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;
size_t m_id;
```

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

10.1.3 The serialize operator

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

```
bool operator()( s11nlite::node_type & node ) const
{
    typedef s11nlite::node_traits TR;
    TR::class_name( node, "MyType" ); // critical, but see below!
    TR::set( node, "id", m_id );
    using namespace s11nlite;
    serialize_subnode( node, "string_list", slist );
    serialize_subnode( node, "child", childs );
    serialize_subnode( node, "int_to_str_map", istrmap );
    serialize_subnode( node, "dbl_to_str_map", dstrmap );
    return true;
}
```

The class name for a registered monomorphic Serializable types can be fetched by calling `::classname<T>()`. In fact, SAM (section 15) does this for you, and the `class_name()` call can technically be left out for monomorphic types. It is probably a good idea to go ahead and include it, for the sake of clarity and pedantic correctness.

10.1.4 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.erase( istrmap.begin(), istrmap.end() );
    dstrmap.erase( dstrmap.begin(), dstrmap.end() );
    slist.erase( slist.begin(), slist.end() );
    acme::free_list_entries( this->childs ); // deletes the pointers
    //////////////////// now get our data:
    typedef s11nlite::node_traits TR;
    this->m_id = TR::get( node, "id", m_id );
    using namespace s11nlite;
    deserialize_subnode( node, "string_list", slist );
    deserialize_subnode( node, "child", childs );
    deserialize_subnode( node, "int_to_str_map", istrmap );
    deserialize_subnode( node, "dbl_to_str_map", dstrmap );
    return true;
```

```
}
```

A note about cleaning up *before* deserialization:

In practice these checks are normally not necessary. s11n never, in the normal line of duty, directly calls the deserialize operator more than one time for any given Serializable: it calls the operator one time directly after instantiating the object. 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.

10.1.5 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 11, and for a quick example see 8.

10.1.6 Done! Your object is now a 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" );
```

(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, without having to know anything but the other libraries' base-most types.

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

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

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

10.2 Gary's code

One of s11n's early-adopters, Gary Boone, contacted me in early 2004 about how to go about adding s11n support to his project. For starters, he had a simple structure (described below). On the surface, the problem appears to be non-trivial, but this is only when viewing the code through the lense of traditional C++ techniques...

Let us repeat the s11n mantra (well, one of several[37]):

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

[37] Trivia note: The banner label on the s11n web site rotates through s11n's list of official mantra, and 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.

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 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 mails 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.

The code below has been updated from Gary's original to accomodate changes in the core library, but it is essentially the same as his original post.

In some places i have added descriptive or background information, marked like so:

[editorial: ....]

10.2.1 *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[38].]

> ...
>
> 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[39] {
```

[38] That text was written some time in the 0.7 or 0.8 cycle, about a year ago (today == 18 April 2005). i still believe that (a) the full limits and implications of the library are not yet fully understood and (b) it really is that flexible. :)

[39] Gary is credited with coming up with the MyType_s11n naming scheme, and it now appears regularly in other s11n client trees.

```
    // 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.
    ************************************/
    template <typename NodeType>
    bool operator()( NodeType & dest, const elem_t & src ) const40 {
        typedef s11n::node_traits<NodeType> TR;
        TR::class_name( dest, "elem_t");
        TR::set( dest, "i", src.index);
        TR::set( dest, "v", src.value);
        return true;
    }
    /************************************
    // 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.
    ************************************/
    template <typename NodeType>
    bool operator()( const NodeType & src, elem_t & dest ) const {
        typedef s11n::node_traits<NodeType> TR;
        dest.index = TR::get( src, "i", -1);
        dest.value = TR::get( src, "v", 0.0);
        return true;
    }
};
```

[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 const/non-const-ambiguity was 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 equation. 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 an existing algoritm. As a general rule, algorithms are slightly easier to test than proxies early on in 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.net/s11n/reg_s11n_traits.hpp>
```

[40] 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* - it may not deviate from that shown here. The constness of the operator itself is not defined by s11n conventions.

[editorial: After this registration, `elem_t_s11n` is now *the* official delegate for *all* de/serialize operations involving `elem_t`. 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 5.4 for an explanation of why taking such an action is considered *Poor Form*!]

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:

```
s11nlite::data_node * node = s11nlite::load_node( "somefile.elem" );
elem_t e;
bool worked = s11nlite::deserialize( *node, e );
delete node;
```

[editorial: that last example basically "cannot fail" unless `elem_t`'s deserialize implementation *wants* it to, e.g. if it gets out-of-range/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 `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 analyzed on a case-by-case basis. In practice this problem is mainly theoretical/academic in nature. 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!

Note that with the various container proxies which ship with s11n, Gary's `elem_t` type can take part in container serialization, such as in a `map<string,elem_t>`
or `list<elem_t>`. There is no separate "serialize container of `elem_t`" operation, as the generic list/map algorithms inherently handle any and all Serializables:

```
typedef std::map<std::string,elem_t> MapT;
MapT mymap;
... populate mymap ...
s11nlite::save( mymap, "myfile.s11n" );
```

11 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, is far, far superior is many ways (the only down-side being it's verbosity). i like to call them...

11.1 "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.hpp’
```

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 MyType
#define S11N_TYPE_NAME "MyType"
#define S11N_SERIALIZE_FUNCTOR MyType_s11n
#include <s11n.net/s11n/reg_s11n_traits.hpp>
#define S11N_TYPE MyOtherType
#define S11N_TYPE_NAME "MyOtherType"
#define S11N_SERIALIZE_FUNCTOR MyOtherType_s11n
#include <s11n.net/s11n/reg_s11n_traits.hpp>
```

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 sets of 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 macros.
- 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 it cannot collide with those used by other registerers.
- As they are implemented in ”real header code”, they are completely immune to the limitations of macros, and simply *much* easier to maintain.
- This approach does ALL necessary registration, including classloader registration (could not be reliably done via the macro approach, due to ODR-violation possibilities).
- 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: unlike conventional macros, 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 implications are still not fully appreciated/understood.

11.2 General: Interface Types

All of s11n's activity is ”keyed” to a type's Interface Type. This is used for a number of internal mechanisms, far too detailed to even properly *summarize* here. A InterfaceType 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 InterfaceType/SerializableType parameters.

(See, it's difficult to describe!)

In most usage using InterfaceTypes 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 InterfaceType to subtypes, for *most purposes*, that looks like this*:*

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

There are valid cases where registering both AType and BType as 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 (or non-existant, for most practical purposes), 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.

11.3 Choosing class names when registering

s11n does not care what class names you use. We could call, e.g. `std::map<string,string>` "fred" and the end effect is the same. In fact, we could also call the pair type contained in that map "fred" - *without getting a collision* - because it uses a different classloader than the map (because they have different InterfaceTypes, as described in section 11.2).

The important thing is that we are consistent with class names. Once we change them, any older data will not be loadable via the classloader unless we explicitely alias the type names via the factory-related API.

By convention, s11n uses a class' C++ name, stripped of spaces and any const and pointer parts. The "noise" parts are, it turns out, *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`) or a (`std::string *`) the type name s11n uses will be "std::string" (or even "string") for *both* of them, and the context of the de/serialization determines whether we need to dynamically allocate pointers or not. It is, of course, up to client code to deallocate any pointers created this way. For example, when deserializing a `list<string*>`, the client must free the list entries. (Tip: see `free_list_entries()` for a simple, generic way to accomplish this for list-type containers.)

11.4 Registering Interface Types supporting serialization operators

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_TYPE_NAME "ASerType"
#include <s11n.net/s11n/reg_s11n_traits.hpp>
```

The registration of a subtype of ASerType looks like:

```
#define S11N_BASE_TYPE ASerType
#define S11N_TYPE BSerType
#define S11N_TYPE_NAME "BSerType"
#include <s11n.net/s11n/reg_s11n_traits.hpp>
```

The `S11N_xxx` macros are `#undef`'ed by the registration code, so client code need not do so, and may register several classes in t a row by by simple re-defining them before including the supermacro code.

11.5 Registering types which implement a custom Serializable interface

If a class implements two serialization functions, but does not use `operator()` overloads, the process is simply a minor extension of the default case described in the previous section. We must do two things:

First, define a functor which, in its Serialization Operators, forwards the call to MyType's serialization interface. An example of such a functor:

```
struct MyType_s11n {
// note that the proxy class name is unimportant:  Gary Boone came up with the XXX_s11n convention
i adopted it
    template <typename NodeType>
    bool operator()( NodeType & dest, const MyType & src ) const {
        return src.local_serialize_function( node );
    }
    template <typename NodeType>
    bool operator()( const NodeType & dest, MyType & src ) const {
        return src.local_deserialize_function( node );
    }
};
```

Second, before including the registration supermacro as shown in the previous section, simply add one or both of these defines:

```
#define S11N_SERIALIZE_FUNCTOR MyType_s11n
#define S11N_DESERIALIZE_FUNCTOR MyType_s11n // OPTIONAL: defaults to S11N_SERIALIZE_FUNCTOR
```

The second functor is only necessary if you define *separate functor classes* for de/serialization operations. In the vast majority of casses one proxy handles both de/serialize operations, so the second macro need not be set.

That's it - you're done telling s11n how to talk to your local serialization API. Now calls to `s11n::de/serialize()` will end up routing through the `local_de/serialize_function()` API.

11.6 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* (*not* proxy) type must be registered with appropriate classloader: monomorphs register with their own, as to Interface/Base-most Types, and subclasses register with their Interface Type's classloader.
- `s11n_traits<ProxiedType>::class_name()` should return the class name which s11n will use for the type. For monomorphs the library can figure this out on its own, but needs help with polymorphic type names.
- An `s11n_traits<>` specialization installed (section 6.2).

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

```
#define S11N_TYPE ASerType
#define S11N_TYPE_NAME "ASerType"
#define S11N_SERIALIZE_FUNCTOR ASerType_s11n
// optional:  #define S11N_DESERIALIZE_FUNCTOR ASertType_des11n
// DESERIALIZE defaults to the SERIALIZE functor, which works fine for most cases.
#include <s11n.net/s11n/reg_s11n_traits.hpp>
```

Note that the names of the de/serialize functors shown here are arbitrary: you'll need to use the name(s) of *your* proxy type(s).
This is repeated for each proxy/type combination you wish to register. The macros used by `reg_s11n_traits.hpp` are temporary, and `#undef`'d when it is included.
There are other optional macros to define for that header: see `reg_s11n_traits.hpp` 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_TYPE_NAME "BSerType"
#include <s11n.net/s11n/reg_s11n_traits.hpp>
```

Without the need to specify the functor name - it is inherited from the type set in `S11N_BASE_TYPE`.

11.7 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, only it's *interface* declaration.)

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 the sloppiest, as is ties type T very closely to libs11n.
- The implementation file(s) making the operation. (Be careful to avoid undue duplication of macro calls, 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.hpp` and `T_s11n.hpp`, with the latter handling s11n registration. This is probably the cleanest solution for non-trivial projects, and is generally the approach taken by s11n's author.

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

11.7.1 Hand-implementing the macro code (IMPORTANT)

The traditional (pre-0.8.x) registration macros are conveniences for handling common cases. They cannot handle all cases, mainly because C macros are so limited. The newer supermacro technique is far superior, and highly preferred.

That said, whenever these docs refer to calling a certain macro, what they *really* imply is: include code which is functionally similar to that generated by the macro. This code can be hand-written (and may need to be for some unusual cases), generated via a script, or whatever. In any case, it must be available when s11n needs it, as described above.

12 Proxies, functors and algorithms

TODO: REWRITE FOR 1.1

s11n's proxying feature is probably it's most powerful capability. s11n's core uses it to proxy the core de/serialize calls between, e.g. `FooClass::save_state()` and `OtherClass::operator()`.

Note that any non-serializable type which s11n proxies is actually a Serializable for all purposes in s11n. Thus, when these docs refer to a Serializable type, they also imply any proxied types. The *proxies*, on the other hand, are not technically Serializables.

How to register a type as a proxy is explained in section 11.6.

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

```
<s11n.net/s11n/data_node_functor.hpp>
<s11n.net/s11n/data_node_ago.hpp>
<s11n.net/acme/algo.hpp>
<s11n.net/acme/functor.hpp>
```

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!)

It must be stressed there is *nothing at all* special or "sacred" about the algorithms and proxies supplied with this library. That is, clients are free to implement their own proxies and algorithms, completely ignoring any provided by this library. If you want, for example, a particular `list<T>` specialization to have a special proxy when you spent all of last night coding, that can be done.

12.1 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_TYPE_NAME "MyType"
#define S11N_SERIALIZE_FUNCTOR serialize_proxy
// #define S11N_DESERIALIZE_FUNCTOR deserialize_proxy
// ^^^^ not required unless noted by the proxy's docs.
#include <s11n.net/s11n/reg_s11n_traits.hpp>
```

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_serializable_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!*

12.1.1 Arbitrary Streamable types: `s11n::streamable_type_serialization_proxy`

This proxy can handle any Streamable type, treating it as a single Serializable object. Thus an int or float will be stored in it's own node. While this is definitely 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 Streamable types they wish.

12.1.2 Arbitrary list/vector types: `s11n::list::list_serializable_proxy`

This flexible proxy can handle any type of list/vector *containing Serializables*. It handles, e.g. `list<int>` and `vector<string*>`, or `list<pair<string,double*>>`, provided the internally-contained parts (like the pair) are Serializable. Remember, the basic PODs are inherently handled, so there is need to register the contained-in-list type for those or `std::string`.

Trivia:

> The source code for this function 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 difficult cases very staightforward to implement in one function.

12.1.3 Streamable maps: `s11n::map::streamable_map_serializable_proxy`

This proxy can serialize any `std::map`-compliant type which contains Streamable types.

12.1.4 Arbitrary maps: `s11n::map_serializable_proxy`

Like `list_serializable_proxy`, this type can handle maps containing any pointer or reference type which is itself a Serializable.

12.1.5 Arbitrary pairs: `s11n::map::pair_serializable_proxy`

Like `list_serializable_proxy`, this type can handle pairs containing any pointer or reference type which is itself a Serializable.

12.2 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. 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**
acme::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.
find_child_by_name()	Finds a sub-node of a node using it's name as a search criteria.
child_pointer_deep_copier	Deep-copies a list of pointers. Not polymorphic.
data_node_child_[de]serializer	De/serialize list<Serializable*>.
acme::object_deleter	Use with std::for_each(), to generically deallocate objects.
acme::pointer_cleaner	Essentially a poor-man's multi-pointer auto_ptr.
s11n::map::de/serialize_streamable_map()	Do just that. Supports any map containing only i/ostreamable types.
s11n::map::de/serialize_[map/list/pair]()	De/serialize maps/pairs of Serializables.
s11n::list::de/serialize_streamable_list()	Ditto, for list/vector types.
acme::object_reference_wrapper	Allows referring to an object as if it was a reference, regardless of it's pointe
acme::pair_entry_deallocator	Generically deallocates pointer elements in a pair<X[*],Y[*]>.
abstract_creator	A weird type to allow consolidation of some algos regardess of argument poi

13 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 is 100% code-independent of it. In s11nlite only one Serializer is used, so this section does not go into detail about how to select one manually. See the sources (`data_node_{serialize,io,format}.hpp`) and `s11nlite::create_serializer( string )` for full details.

13.1 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 DOM conventions.
- Nodes may represent any Serializable type, with all that that implies, or "raw" data nodes (without type meta-information).
- Nodes may contain an arbitrary number of child nodes.
- Nodes must have a name meeting the criteria specified in section 5.3. The name need not be unique within that branch of the tree.
- Nodes must have an "implementation class name" set - 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 9.4 work this way). Use `s11n::node_traits<NodeType>::class_name(node)` to set the class name.
- 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 5.3 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.

13.1.1 File extensions

File extensions are irrelevant for the library - client files may be named however clients wish. 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.)

13.1.2 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 - to the order of 10% of a file's size for data sets containing lots of small data (e.g. integers).

13.1.3 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 shown 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::cl::classload<SomeSerializerInterfaceType>()`).

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 knows 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 own 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.hpp` for samples of how to add new Serializers to the framework.

Versions 0.9.7 and higher support a special cookie which can be used to load arbitrary Serializers without having to pre-register them. If the first line of a file looks like this:

```
#s11n::io::serializer ClassName
```

then ClassName is classloaded as a Serializer (a subtype of `s11n::io::data_node_serializer<>`) and, if successful, that object is used to parse the remainder of the stream.

13.2 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, with the exception that the `expat_serializer` is XML-standards compliant, and is very unforgiving about things like numeric node names.

As of version 0.9.14, the available Serializers are shipped as DLLs, not linked in directly with the library. s11nlite auto-loads the "known" Serializers (those shown below) at startup, but clients will have to load their own DLLs if they provide any. See `s11nlite.cpp:s11nlite_init()` for a sample implementation which loads a known list of DLLs.

13.2.1 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 size - failing to do so effectively corrupts the data.
- Node/key/class names are limited to 255 characters.
- Property data is "limited" to 4GB per property.

Sample:

51191011[41]

```
f108somenode06NoClasse101a0003foo...
```

13.2.2 expatxml

Serializer class: `s11n::io::expat_serializer`

This Serializer, added in version 0.9.2, uses libexpat[42] and is only enabled if the build process finds libexpat on your system. It is grammatically similar to funxml (section 13.2.4), but "should" be more robust because it uses a well-established XML parser. Additionally, it handles self-closing nodes, something which funxml does not do.

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.
- Not thread-safe: it is not safe to read from more than one of these objects at a time, e.g. in a client/server environment.
- XML standards compliant, which means it does not tolerate extensions supported by the other s11n XML formats, like numeric node names.

Sample:

```
<!DOCTYPE s11n::io::expat_serializer>
<nodename class="SomeClass">

    <property_name>property value</property_name>
    <prop2>value</prop2>
    <empty_property/>
    <empty_class class="Foo"/>

</nodename>
```

[41] "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).

[42] http://expat.sourceforge.net

13.2.3 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 libFunUtil, as it has been used in the QUB project since mid-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 common uses. It also has the benefit of being one of the most human-readable/editable of the formats.

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[43]:

- 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).

13.2.4 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.
- To help support the various container serialization functions (section 9.4), 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 ... />`.

Sample:

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

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

</nodename>
```

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

13.2.5 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 13.2.3) 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)
        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 are 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.

> *Historical speculation*: that might explain why, in STL documentation, they denote iterator begin/end ranges in the form `[B,E)`, where "[" means *inclusive* and ")" means *exclusive*. If the symbols were defined the other way around, such that `(B,E]` had the same meaning as above, emacs's paren-matching and indention modes would get out of sync, which would *most certainly* have frustrated the designers of the STL. :) Even if that is not the case - which it is probably is not - the paren serializer *does* explicitely have this escaping behaviour to accomodate emacs. Yeah, i know that a *real*, *die-hard*, *lisp-loving* emacs user [with way too much extra energy] would have simply implemented `paren-serializer-mode`... and probably would have implemented the C++-side serializer class on top of it. And it would work, too, because emacs is just cool that way. But i haven't got *that* much energy, and thus the above-mentioned backslash hack was introduced.

13.2.6 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. (Historical note: i wrote this Serializer in October, 2003, and have never once used the CDATA feature outside of test cases.)

Known limitations:

- See the caveats/limitations notes in section 13.2.4. 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 in storing each node's `impl_class()`.

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>
```

13.2.7 wesnoth

Serializer class: `s11n::io::wesnoth_serializer`

"wesnoth" is a simple text format based off of the custom data format used in the game *The Battle for Wesnoth* (www.wesnoth.org).

Known limitations:

- New (added in 0.9.14) and not well-tested.
- Does not yet properly support multi-line strings as property data. (At least, it's not tested.)

Sample:

```
#s11n::io::wesnoth_serializer
[s11nlite_config=s11n::data_node]
GenericWorkspace_size=1066x858
s11nbrowser_size=914x560
serializer_class=wesnoth
[/s11nlite_config]
```

13.3 Tricks

13.3.1 Using a specific Serializer

Easy: simply pick the Serializer class you would like 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.

13.3.2 Selecting a Serializer class in s11nlite

See `create_serializer(string)`, which takes a classname and can load any registered subclass of `s11nlite::serializer_b`
Alternately, set the framework's default serializer type by calling `s11nlite::serializer_class(string)`. As of 1.1, this setting is no longer automatically persistent across all s11n clients: client applications must either set this at some point or rely on the compiled-in default (which will be some built-in Serializer, but *which one* is not specified by s11nlite's interface).

13.3.3 Multiplexing Serializers

This has never been done, but it seems reasonable:

If you'd like to 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 simply 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. i can personally see little benefit in doing so, however (maybe a more creative soul can find a clever use for it, though... e.g. protocol-within-protocol wrapping for an RPC channel).

14 `class_name()` & friends

TODO: rewrite for 1.1

Once upon a time - the first few months of s11n's development - s11n developed a rather interesting trick for reliably 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, well-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 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".

14.1 `class_name()`

> **Note**: in older s11n code we had an `impl_class()` is function in some contexts. That was identical to `class_name()`, but is long-since deprecated. The documentation may still refer to `impl_class()` in some cases, but these can be safely understood to mean `class_name()`.

For s11n, a node's metatype class name is significant at the following points:

1. When serializing an object, the node it is stored in should have it's `class_name()` set to the object's class name. This is trivial 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 `class_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 `class_name()` *after* calling the inherited behaviour, to avoid that the parent type overwrite the `class_name()` of the subtype.
Note that Serializable Proxies need to set 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 InterfaceType, as in this code:
   `InterfaceType * b = s11nlite::deserialize<InterfaceType>( somenode );`
   s11n asks the InterfaceType's classloader (e.g. `s11n::cl::classloader<InterfaceType>()`[44]) for an object of type mapped to the name stored in `node_traits<NodeType>class_name(somenode)`. The classloader, ideally, has a subtype of InterfaceType registered with that name (or it is InterfaceType'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 *InterfaceType'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, targetser )`) then the class name 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 a complementary deserialize algorithm or functor which understand the "data layout" within the node. e.g. 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. This is not to say that the string must be the same as a class' name: you may of course use numeric strings.

Hopefully the significance of a node's class name 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 the development mailing list 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. std::__gcc_blahblah_internal<Foo *,std::alocator<Foo> .....>).

14.2 `classname<>()`, `class_name<>`, `name_type.hpp`

TODO: rewrite for 1.1

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.net/name_type/class_name.hpp>
// ...  declare or forward-declare MyType ...
CLASS_NAME(MyType);
```

[44] This is not entirely true: clients may customize the default factory handling, but doing so is currently outside the scope of this document.

Metod #2: (*highly preferred*)

```
#include <s11n.net/name_type/name_type.hpp>
#define NAME_TYPE mynamespace::MyType< TemplatizedType >
#define TYPE_NAME ”mynamespace::MyType<TemplatizedType>”
#include <s11n.net/s11n/name_type.hpp>
```

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 helpful 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. That is there should be no two types registered with the same name for class_loader<T>, but two types may be registered with the same names for different classloaders, class_loader<X> and class_loader<Y>.
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 preferred convenience form:

```
::classname<MyType>()
```

Sounds pretty simple, right? If Method #2 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.hpp.
- 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 generally does not provide 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 instead returns typeid<T>::name() and sends a warning to stderr.
- In the case of class_name<T*>, the pointer part of the type is *not* represented in the name. i.e., class_name<T*>::name() returns ”T”. This behaviour has a long set of justifications. Suffice it to say that leaving it off simplifies significant parts of s11n's internals, and also makes s11n more flexible and more efficient 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_serializable_proxy and s11n::pair_serializable_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. 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., normally meaning 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 registering `class_name<MyType>` in `MyType.hpp` is a guaranteed way to give all users of MyType a proper `class_name<MyType>` specialization without risking ODR violations.
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 in general preferable to do so, mainly for maintenance reasons, but arguably also for a style point or two.
- `::classname<T>()` lives in the global namespace. This is primarily to ease typing, because this function is called quite often, particularly in Serializable Proxy code where a generic proxy needs to set the proper name of a type it is asked to serialize. (Remember, `impl_class()` is irrelevant once the class is loaded, so the deserialize operators never have to deal with it.)

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 specializations conform to the above interface and are consistent (changes - 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.

15 SAM: Serialization 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. There are cases - especially when serializing class templates - when writing a SAM is just what is needed, however.

Achtung #2: There is a fine line, and indeed some overlap, between certain responsibilities of SAM and those of `s11n_traits<>`... but the line isn't well-defined and the small overlap is actually a flexibility benefit (e.g. where is a node's `class_name()` set?). In effect, `s11n_traits<>` provides the *public* interface for API marshaling and SAM provides the *s11n-internal* interface. Traits and SAM also each have some very distinct responsibilities, and consolidating them into one type is not planned.

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: Serialization API Marshaling[45] layer

[45] Note that both "marshaling" and "marshalling" are correct spellings of this word. s11n uses the single-l variant because `ispell` told me that was correct ;).

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 brains suit you just fine. If that is indeed the case, keep reading to learn the Truth about SAM...

15.1 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 these small details:

- SAM's two API functions and their conventions (which are identical to those of s11n's core de/serialize functions).
- `node_traits` (section 6.1), and only a small portion is used internally.
- `s11n_traits` (section 6.2).

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 - looking over the code now i'd guess that, if we don't count the `[de]serialize_subnode()` convenience funtions, it's *less than 30* actual code lines(!!!).

SAM defines the interface between s11n's core and the world of client-side code. The following code reveals the *entire* client-to-core communication interface:

```
template <typename SerializableT>
struct s11n_api_marshaler {

     typedef SerializableT serializable_type;
     template <typename NodeType>
     static bool serialize( NodeType &dest, const serializable_type & src );
     template <typename NodeType>
     static bool deserialize( const NodeType & src, serializable_type & dest );

};
```

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 overloads of `operator()`. And (there's another, lamer reason) the `operator()`- style interface can sometimes cause ambiguity errors, so it needs to be avoided here.

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

It is important to understand *how* s11n "selects" a SAM specialization: by the *type* argument passed as a SerializableType template parameter. 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...

15.2 SAM's place in the API calling chain (and other important notes)

After client code initiates a de/serialization operation, the process goes something like this:

1. s11n passes off the call to to `s11n_api_marshaler<T>::[de]serialize(node,obj)`.
2. SAM is now in control of the request. The default SAM implementation simply delegates the request to `s11n_traits<T>::[de]serialize_functor`, as appropriate.
3. SAM eventualy returns to the core, which then passes the results directly back to the user.

In API terms, SAM is *the internal place* to manipulate the marshaling process, e.g. to implement custom API translation. The *public* interface for doing so is by specializing `s11n_traits` for a given type.

As a special case[46], SAM<X *> is single implementation, not intended to be further specialized - see below!

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."

15.2.1 More about SAM<X*>

A *single* specialization of SAM<X*> does pointer-to-reference argument translation (since its SerializableTypes will be pointer types) and forwards them on to SAM<X> (unless they are 0, in which case it simply returns false - effectively a failed de/serialization attempt). 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 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*>, and the pointer will be dereferenced before passing it to SAM<Y>.

Some coders show a level of distrust for this "feature", but practice has shown that it is 100% non-intrusive, 100% predictable, and allows some tricks which are otherwise difficult to achieve. In fact, code related to this specialization has not needed any maintenance since its initial introduction, a bit more than a year ago - it is a pure background detail.

Client code SHOULD NOT implement any pointer-type specializations of `s11n_api_marshaler<X*>`[47]. Clients MAY implement such specializations, but they're on their own in that case. As it is, 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 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, and related problems could be extremely difficult to track down later.

16 s11n-related utilities

This section list the utility scripts/applications which come with s11n, plus some tools which are known to be useful with s11n but are not shipped with it.

16.1 `s11nconvert`

TODO: not yet ported to 1.1 because the command-line parsing code (which does 80% of s11nconvert's work) went bye-bye in the code re-org.

Sources: `src/client/s11nconvert/main_dn.cpp`

Installed as `PREFIX/bin/s11nconvert`

`s11nconvert` is a command-line tool to convert data files between the various formats s11n supports.

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

Sample usages:

Re-serialize inputfile.s11n (regardless of 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
```

[46] 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.

[47] ... 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.

Note that zlib/bzip2 input/output compression are supported for *files*, but not when reading/writing from/to standard input/output[48]. You may, of course, use compatible 3rd-party tools, such as gzip and bzip2, to de/compress your s11n data. Also note that compression is only supported if the `zfstream` supplemental library supports it.

16.2 `s11nbrowser`

TODO: not yet ported to 1.1.

s11nbrowser is a Qt-based GUI application for reading arbitrary data saved with s11nlite. It is not shipped as part of s11n, but is distributed as a separate application, available from:

```
http://s11n.net/s11nbrowser/
```

17 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. The more complex features are documented mainly in the source files which implement them. Some features, e.g. those implemented via macro-based code generation, are not processed by the API doc generator, so the implementation headers are the only complete source of info for most macro-based features. There are cases where some of the macro-generated code must be hand-implemented (or externally generated), so it is useful to understand exactly what the macros do.

17.1 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 9.4). 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::node_type node;
s11n::map::serialize_streamable_map( node, theConfig );
s11nlite::save_node( node, "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 `s11n::data_node` as a primitive config object.

If the Config object is a Serializable object (or a proxied one) it becomes even simpler: simply use the `save/load()` or `de/serialize()` functions directly on the object. Hint: std::maps are proxied by default, so doing so requires no client-side code.

[48] Sorry, we don't have an in-memory de/compressing streambuffer for these.

17.2 Saving application-wide state and Singletons

It is sometimes useful to be able to serialize the state of an application though we have no specific object which holds all application data. This can be handled by defining a simple Serializable which saves and loads all global data via whatever accessors are available for the data. The same approach can be used for Singletons, which we would not normally be able to dynamically load via deserialization due to their Singletonness. An example of how to set this up:

```
struct myapp_s11n // our "placeholder" Serializable type
{

    template <typename NodeT>
    bool operator()( NodeT & node ) const // Serialize operator
    {
        typedef s11n::node_traits<NodeT> TR;
        TR::class_name( node, "myapp_s11n" );
        ... use algos to save app's shared state ...
        return true;
    }
    template <typename NodeT>
    bool operator()( const NodeT & node ) // Deserialize operator
    {
        ... use algos to restore app's shared state ...
        return true;
    }

};
```

Then register it as a Serializable, which is simpler than for most proxy cases because our "proxy" is actually a Serializable implementing the so-called Default Serializable Interface:

```
#define S11N_TYPE myapp_s11n
#define S11N_NAME "myapp_s11n"
#include <s11n.net/s11n/reg_serializable.hpp>
```

To save application state, we simply need:

```
myapp_s11n state;
s11nlite::save( state, "somefile.s11n" );
```

Note that passing an unnamed temporary `myapp_s11n` to `save()` would not compile because non-const temporaries are not legal function arguments in C++.

To load our app state we can take a couple of different approaches, but the most straightforward is probablye:

```
myapp_s11n * state = s11nlite::load_serializable<myapp_s11n>( "somefile.s11n" );
delete( state ); // no longer needed - it modified the global state for us.
```

Or, if you want to get fancy, perhaps something like:

```
{ // create a scope to contain the following auto_ptr<> object...
    std::auto_ptr<myapp_s11n> ap(
        s11nlite::load_serializable<myapp_s11n>( "somefile.s11n" )
    );
    if( ! ap.get() ) { ... load failed ... }
}
```

17.3 "casting" Serializables with `s11n_cast()`

Serializable containers of "approximately compatible" types can easily be "cast" to one another, e.g. `list<int>` can be "cast" to a `vector<int>`, or even a `list<int>` to a `vector<double*>`. What exactly constitutes "approximately compatible" essentially boils down to this: the two types must have the same or compatible s11n proxies installed.

Assuming we have registered the appropriate types, the following code will convert a list to a vector, as long as the types contained in the list can be converted to the appopriate type:

The *hard* way:

```
s11nlite::node_type n;
s11nlite::serialize( n, mylist ); // reminder:  might fail
s11nlite::deserialize( n, myvector ); // reminder:  might fail
```

Or, the *slightly-less-difficult* way:

```
s11nlite::node_type n;
bool worked = s11nlite::serialize( n, mylist ) && s11nlite::deserialize( n, myvector );
```

Or, the *easy* way:

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

Done!

Reminder: if this operation 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.

Reminder #2: it is important to remember that only types which use compatible de/serialization algorithms may be `s11n_cast()` to each other. The reason is simply that the de/serialize operators of each type are used for the "casting", and they need to be able to understand each other in order to transfer the object's state.

17.4 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. To be certain that the proper classloader is used, you should explicitely pass the templated type, using the base-most Serializable type of the hierarchy. When cloning *monomorphs* this template typing is not an issue (unless the type may one day become a polymorph, in which case *not* explicitely specifying the template parameter is potentially bug in waiting).

17.5 zlib & bz2lib support

s11n supports file de/compression using zlib and bz2lib via the zfstream sub-library. However, in the interest of data file portability/reusability, *file compression is off by default.* Use `zfstream::compression_policy()` to set the library's default file compression policy (defined in `<s11n.net/zfstream/zfstream.hpp>`).

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 `zfstream::compression_policy()`: see `zfstream::get_istream()` and `get_ostream()` if you'd like your client code to do the same. Note that compression is not supported for arbitrary streams, only for files. Sorry about that - we don't have in-memory de/compressor streambuffer implementations, only file-based ones (if you want to write one, PLEASE DO! :).

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

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 = (zfstream::compression_policy( zfstream::GZipCompression ),0);
```

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

17.6 Using multiple data formats (Serializers)

It is possible, and easy, to use multiple Serializers, from within in one application. s11nlite likes to hide this detail from us, but allows us to set the default Serializer class and load Serializers by class name at runtime.

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: see `serializers.hpp` for samples of how this is done.)

Saving data to a stream necessarily requires that the user specify a format - that is, client code must explicitely select its 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 (section 16.1).

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. Theoretically, it might be easily achievable if... no, we won't go there.

17.7 Loading Serializables dynamically via DLLs

TODO: REWRITE for 1.1 - DLL support does not currently exist in the 1.1 tree.

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 its customization is, so far, undocumented: see lib/cl/src/cllite.hpp.

17.8 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 definitely 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.

17.9 Containers of const objects

When serializing containers of const objects, we need to do some special-case handling during deserialization. To make a very short example, let's assume that our class contains a list which we would like to serialize:

```
typdef std::list<const MyType *> ListT;
```

That will *serialize* just fine, but *deserialization* will fail at compile-time because the deserialization algorithm of MyType is non-const, and thus may not modify the object it needs to modify. It is an inherent property of *De*serializables that they may not be const, just as it is an inherent property of Serializables that they must[49] be const.

In this case we need to apply the layer-of-indirection rule. One straightforward approach is, in our deserialize operator, to deserialize the list to a temporary container of `list<MyType*>`, then copy or move the pointers into your ListT, like so:

```
typedef std::list<MyType *> TempT;
TempT tmplist;
if( s11n::deserialize( mynode, tmplist ) ) {
```

[49]Well, "should" be const. Most serialization libraries do place const requirements on serializable types.

```
    ... copy/move tmplist's contents to our member list ...
}
```

We must of course be careful with the pointer ownership: `tmplist` owns the pointers initially, and we will need to move that ownership to wherever is appropriate for our application.

Note that it is theoretically possible to add a simple wrapper which handles this const-related handling for a certain class of container (e.g. lists or maps), such that we could do something like:

```
deserialize_list_of_consts( mynode, mylist );
```

The function would need to internally strip out constness from `ListT::value_type`, so it would be some interesting template meta-code, but i believe it could be done with little effort.

17.10 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.

While the toc project itself appears inactive, toc is *continually* under development in the s11n tree and other trees i work on.

18 *Absolute No-no*'s (Worst Practices) for s11n[lite] client code

This section, added in version 0.9.17, covers some "no-no's" for the s11n framework. That is, things which are often easy to do but should not be done. They are here because, well, because i've done them more than once and want to spread the word ;).

Please note that the subsection titles below all start with the words *do not* and end with an exclamation point!

18.1 *Do not* change the name of a passed-in data node!

`node_traits<>::name(string)` is used to set the name of a node. This name is used by Serializers to, e.g. name XML nodes:

```
<nodename s11n_class="MyClassName">...</nodename>
```

As a blanket rule:

> **No code must ever change the name of a node which is *passed to it*. Code may freely change the names of *nodes which it creates*.**

Also keep in mind that if you want to support the widest variety of data fomats, you should follow the standard data node naming conventions covered in section 5.3.

An example of this no-no:

```
bool my_algo( s11nlite::node_type & dest, const my_type & src )
{
    typedef s11nlite::node_traits NTR;
    // NONO: NTR::name(dest,"whatever");
    // Never change the name of a node passed to us.
    // The following is Perfectly Acceptable:
    s11nlite::node_type * child = NTR::create();
    NTR::name(*child, "foo" );
```

```
    // alternately:
    // child = NTR::create(”foo”);
    NTR::children(dest).push_back(child);
    // or create, name, and reparent in one step:
    // child = & s11n::create_child( dest, ”foo” );
}
```

The reason for not changing the name is essentially this: when building up a tree of nodes, the easiest way to remember nodes (for s11n's purposes) is normally to name them. When a function names a node during *serialization*, the matching *deserialization* algorithm will rightfully expect to be able to find the named node(s). When it cannot find the named node(s), deserialization will likely fail (this depends on the algorithm and data structure, but generally this would indicate a failure). To be perfectly clear: this means that *serialization* is likely to pass by *without error* (in fact, it's almost guaranteed to), but *deserialization* will likely fail (again, "it depends", but it *should* fail).

18.2 *Do not* use a single Data Node for multiple purposes!

See also section 21.2.

Never do something like the following:

```
s11nlite::serialize( mynode, mylist );
s11nlite::serialize( mynode, myotherlist );
```

We've just serialized two lists into the same data node (`mynode`). Unless you specifically design algorithms/proxies to handle this, the results are *undefined*. Likewise, the following is a related no-no:

```
s11nlite::node_traits::set( mynode, ”myproperty”, myval );
s11nlite::serialize( mynode, myotherlist );
```

Again, we've used `mynode` for two complete different things: storing a property *and* list. If the property is not hosed by the list serialization algorithm then the extra property in the node may very well confuse the deserialization algorithm! Again: *undefined behaviour*.

Doing these types of things will quite possibly cause a "logical failure" during deserialization. That is, the de/serialization will work, in and of itself, but the results will not be what are semantically expected (but are, indeed, exactly what s11n was told to do).

That leads us to a related no-no...

18.3 *Do not* re-assign a reference returned by `s11n::create_child()`!

Never re-use a reference returned from `s11n::create_child()` as the target of an assignment to another `create_child()` call. In other words, don't do this:

```
s11nlite::node_type & n = s11n::create_child( mynode, ”subnode” );
... serialize something to n ...
... Let’s re-use n for another subnode ...
n = s11n::create_child( mynode, ”othersubnode” ); // Doh! Just re-assigned the ”subnode” node!
```

That's almost certainly not what's intended. What we probably meant to do was:

```
s11nlite::node_type * n = & s11n::create_child( mynode, ”subnode” );
... serialize something to n ...
n = & s11n::create_child( mynode, ”othersubnode” ); // fine
```

(The changes are marked in blue.)

The design reason that `create_child()` returns a reference is because it returns a non-`const` which is not owned by the caller (it belongs to the parent node), and i want the interface to intuitively reflect that the caller does not own the returned object. In general C++ practice, object ownership is never transfered to the caller when a function returns a reference.

18.4 *Do not* use Serializers to implement classical i/ostream operator functionality!

It may be temping to implement classical-style i/ostream operators by using s11n. The core of s11n is i/o ignorant, and using it directly from within your i/o operators is possible, but potentially tedious. The `s11n::io` namespace provides classes which use s11n's conventions to provide a streams-based i/o layer. s11nlite provides a binding between the `s11n::io` layer and the core layer. It may be tempting to bypass s11nlite and use the `s11n::io` layer from your i/o operators. That is unlikely to work, largely because of the workflow Serializers are designed to follow. Serializers rely on a strict sequence of events which says, "read/write *one* top-level node from/to *this* stream, then you're done." When using Serializers for arbitrary sequences of i/o operators, the Serializer cannot precisely know when a *root* node begins, and thus get confused. If i/o operations are freely mixed in arbitrary order (as they easily could be when dealing with client-side i/ostream operators), the Serializers aren't smart enough to deal with it, as it's far outside of their scope.

Don't forget: if a type is Streamable (i.e., supports i/ostream operators) then it is *inherently* Serializable: if it wants to be treated as a full-fledged Serializable, instead of as a POD, a proxy needs to be installed, such as `s11n::streamable_type_serialization_proxy`. See the various `pod_XXX.hpp` proxy-installation headers for examples of how this is done.

18.5 *Do not* register a type as it's own proxy!

Okay, this is not specifically a "do not", but there are good reasons not to do this. Do what? Do this:

```
#define S11N_TYPE MyType
#define S11N_TYPE_NAME "MyType"
#define S11N_SERIALIZE_FUNCTOR MyType
#include <s11n.net/s11n/reg_s11n_traits.hpp>
```

Proxy objects are created very often - on each call to a de/serialize operator - then immediately destroyed. Unless your type is extremely cheap to create and copy, do not register a type as its own proxy. The default proxies are cheap by design, and have no per-instance state.

Aside from that, this type of registration essentially just doesn't make sense, and no use case to date has shown a need for it. It's really one of those dreaded academic/theoretical problems which is unlikely to ever actually show up. But consider yourself warned, nonetheless.

19 Miscellaneous caveats, gotchas, and some things worth knowing

19.1 Serializing class templates

Please see the examples on the s11n web site, which covers this whole process in detail. Fundamentally it is not different from handling any other class, but there are some special considerations which have to be accounted for when registering them.

19.2 Compiling and linking s11n client applications

Use the `libs11n-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 `libs11n-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 the file `/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:

```
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)
...
```

19.3 Cycles and graphs

While i have *never* seen it happen, it is possible that a cyclic de/serializing object will cause an endless loop in the core, which will almost certainly lead to much Grief and Agony on someone's part (probably yours!). Such a problem is almost certainly indicative of mis-understood or incorrect object ownership in the client code. Consider: presumably only an object's owner should serialize that object, and child objects should generally never have more that one parent or owner.

Data Node-based de/serialization (as opposed to Serializable-based) never inherently infinitely loops because Data Node trees simply don't manage the types of relationships which can lead to cycles. In other words, any such endless loops must be coming from client code, or possibly from client-manipulated Data Node trees.

At least one algorithm has been implemented on top of s11n to serializer containers of a graph of client-side objects, but that particular one was proof-of-concept and it can be implemented *much* better that i have. The point being, it *can* be done, but the library current ships with no algorithms to do this. *If you write one, or even a good, generic description of how to implement one, please submit it!*

19.4 Thread Safety

To be perfectly correct, there are no guarantees. i have no practical experience coding in MT environments, and thus it would be a blatant lie if i made *any* sort of guaranty in this area. But i will tell you what i *think* are the facts...

The s11n code "should" be "fairly" thread-safe, with some notable caveats:

First off, no two threads should ever use the same Serializer instance at the same time: each thread must start and finish its de/serialize before another thread does so. Violation of that rule is a blanket no-no.

The following Serializers are believed to be 100% thread-unsafe (or un-thread-safe, if you prefer) in all regards:

- compact_serializer
- simplexml_serializer
- expat_serializer

The Serializers parens, funtxt and funxml have been extensively reworked to use instance-specific internal parsing buffers, as opposed to global data, and are *believed* to be safe in the sense that you may use N instances on N streams from N threads at once. (Let me stress: that is *theory.*)

The lack of thread safety guarantees means that s11n cannot currently be safely used in most network communication contexts, 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).

20 Understanding the costs of deploying s11n

> (Why is this section so far down in the manual, when this info really should be up near the top? Because it goes into quite a lot of technical detail which will only be fully understood once the s11n architecture is understood. It's kind of a chicken-egg scenario.)

Having a generic, widely-useful serialization framework at hand means, for me, saving tens to hundreds of hours of work on other project trees. Literally, every time i add s11n support to a project, after 10 minutes of work i can say, "thank gawd that's over!"

But of course all lazy programmers end up paying somewhere... and this section is about the overall deployment costs of using s11n in client-side code. While it may not be conventional for a library to document this type of thing, i feel compelled to tell it like it is, if only to clear my conscience of all the hype i've been spouting about the library up until the point ;).

To be clear, *all* software has deployment costs associated with it - this is not a detail which is specific to s11n! By "costs" we mean things such as:

- Developer learning time.
- Code refactoring effort (if applicable - s11n can often be included in a project *post facto* with minimal changes to client code).
- Compilation times. This is definitely s11n's sorest point, due to its heavy use of templates. Much work has gone into cutting these down in the 1.1 tree, by being much more selective about what back-end types we automatically create.
- Runtime resources: primarily memory and potentiallly filesystem space.

This section will attempt to address these costs, to give potential users of the library a good idea of what they might be getting themselves into... hopefully before they get into it. We will not provide many hard numbers, but we will give an overview of where one can expect to incure at least some minimal amount of deployment overhead.

For completeness, we really should compare s11n's costs in at least the following contexts:

- The cost of custom-implementing serialization. Even for trivial cases, this has shown to have non-trivial costs.
- Compared to integrating "the average 3rd-party library". This of course varies *widely*, depending on the nature of the client-lib dependency, so a blanket comparison cannot be validly made here.
- Compared to the cost of using an equivalent serialization library.

That last context isn't really fair, because there currently is no such alternative ;). To be fair, though: see `http://boost.org`, and look for Robert Ramey's serialization library, for the only other C++ serialization framework which currently offers *anywhere near* the levels of flexibility and features offered by s11n. i would guess that Robert's library has similar overall deployment costs as s11n, perhaps even slightly lower (but also less feature-full), and of course has the advantage of the *massive* peer-review system that all Boost libraries go through.

While normally we won't go into specifics of s11n vis-a-vis other alternatives, if only because i only use s11n for all of my serialization needs ;), we will attempt to provide an as-object-as-possible overview of the general types of deployment costs.

As with *any* software, the cost of deployment is a cost paid almost entirely by the *clients* of that software (who may also be the software's developers, as in the case of "internal" software). i *personally* feel that s11n has a relatively low cost of deployment, particularly when compared to the alternative of hand-coding serialization support into a library. That said, i would be extremely interested in hearing *your* own experiences and opinions (or hard facts!) about s11n's cost of deployment. Suggestions for lowering *any* aspect of deployment costs are always welcomed. :)

20.1 Learning curve

It would not really be fair for me to comment on this aspect of s11n. As its author, i inherently know how s11n works and how to use it. But i will of course comment on it, otherwise this section would end immediately after this paragraph.

It is my belief that experienced coders who start with the sample code in the s11n source tree and browse through the docs can pick up the library, almost to the point of full profiency, within a day or two (maybe faster, for you especially clever ones out there). It can be understood to the point where one can basically use it in a couple of hours or less, i would think. (*If i am way off here, please let me know!*)

My "experienced guestimate" would say that coders who have posted to the s11n mailing list normally seem to feel comfortable with the architecture after writing 2-3 serializable implementations or serialization algorithms. i can't say how physically long that maps to for beginners - an experienced s11ner can crank out such an implementation in a few minutes in most cases.

Please, please, *please*, if you are just starting out with s11n, *start with the s11nlite API*! See section 2.4 for why.

True masterhood of the library can take time, but how much is unknown and probably unknowable. i will admit that i do not yet fully comprehend all of the *potential* uses, abuses, and tricks implied by the architecture. There's still a lot of room for theory in there, and at least as much room for experimentation. It will be a while before s11n's current model is worn out, i think (i hope!). Exploring those aspects is half of the fun of working on s11n.

There *is* a *lot* of documentation for the library, but that is *not* because it's hard to use. That is, rather, because:

1. As a client-side software user, i refuse to use undocumented libraries, with a strong preference towards *well-documented* libraries (e.g. Qt (`http://www.trolltech.com`) is a good example, as are the libraries available from `http://boost.org`). Being so pedantic on this point, i cannot expect users of my software to give it a second glance if it's not documented, and not to give it a third glance if simple things like pointer ownership aren't documented. You wouldn't believe how much software does not document pointer ownership. Aaarrggg.
2. Experience shows that documenting software helps to find weaknesses in the API. e.g. if something is difficult to document clearly, it's almost certainly difficult to use properly. Holes in the API have often been caught by documenting the related APIs.
3. i enjoy writing about topics which interest me, and s11n obviously interests me.

Users are *not* expected to read the full documentation in order to be able to use the library, but it is hoped that the documentation will be able to answer most or all of their questions, should they need a reference. If they don't, feel free to email us your questions (the address should be at the top of this document).

20.2 Intrusivity (or not)

s11n goes to great pains in order to be as non-intrusive as practical on client code. Clients wishing to support a "conventional" serialization API, where classes derive from some Serializable base type, will of course require some level of hard dependency on s11n. Clients who use s11n's proxy support can, in many cases, add serialization without having to change their core project code *at all* - rather, they simply need to add and register the appropriate proxies . Using the proxy approach can help keep client-side dependencies on s11n down to a handful of places.

20.3 Compilation costs

Yes, i *actually do* have something very negative to say about libs11n: client-side compile times absolutely suck. This was especially true in versions before the mid-0.9 series, and is still a sore point for 1.0.x.

The reasons for the horrible compilation times boil down to:

- We internally create many small template types during compilation to achieve "compile-time polymorphism" and factory registrations for the s11n API. The former is required for s11n's API marshaling technique, amongst other things, and the latter is required for dynamic creation of objects during deserialization[50].
- Compiling template code inherently takes more compiler horsepower than non-template code, especially when advanced features like partial template specialization are used.
- Compiled template code inherently generates much larger object code than non-template code does. This means inherently longer link times, to resolve multiple copies of templates. This also means *significantly* larger object files, which inherently means more i/o is required by both the compiler and linker(s). Whether or not this specific aspect plays a significant build-time role is arguable, and has never been benchmarked, but it is at least worth mentioning and cannot be completely ruled out as a problem point.

In the 1.0 tree, the main culprits for chewing up compile times are the various proxy registrations: it goes overboard and installs many of them in cases where it doesn't need to in order to simplify client-side usage. In the 1.1 tree we have factored out the proxy registrations into as small of units as are practical. This requires a bit more forethought on the developer's part, as he must decide which headers/proxies he needs to include, but the compile-time benefits should be noticeable in the vast majority of client-side cases. At least, it is hoped that they will be more tolerable :/. In practice, the new approach actually seems easier to learn, because the

[50] That's not entirely true as a blanket rule for deserialization, but it is a rule for s11n's implementation. We could ditch the factory layer if we either had no support for polymorphism or required the user to provide us with explicite functions for creating their types. Neither of those are acceptable, of course (though the latter can be legitmately done with s11n's factory, there is normally no need to do so).

new naming conventions normally point the developer to the proxy he needs, whereas the 1.0 tree doesn't have a consistent convention for including existing proxy headers.

Again, my appologies for the slow compiles, but i simply don't see a way around this problem without doing things like build-time code generation, where we could build the s11n-related code one time in a separate module. Code generators are out of the question, as far the s11n core goes, because they is not in-language. That said, clients are free to do whatever code generation they feel they need to. By pre-generating s11n proxies and compiling ALL s11n support into localized object files, is is theoretically possible to shift the compile-time hits to only those modules. Theory, that is, compilicated by the nature of template instantiation rules. If you pull it off, please share with us how you did it.

20.4 Memory/RAM costs.

Here we will focus on the *theoretical* costs of system memory (RAM) *vis-a-vis* serialization via s11n. Filesystem space is not a special concern in the context of s11n, as filesystem limits apply to any code which saves data. That said, s11n's i/o layer does no unusual tricks, using only the standard i/ostreams interfaces, so s11n should not exhibit any sort of "unusual" file access costs. Likewise, it does no unusual memory-related tricks like reimplementing `new` or `delete`, or using custom allocators.

At an abstract level, serializing an object requires that we make a logical copy of the object. This is of course not cheap, even if only because Serializable objects have, by their very nature, some number of data members. In abstract terms, let's naively assume that the copy is twice as large as the original. In concrete terms, this is highly unlikely to be the case: the serialized data of course has its own internal overhead. To understand what this overhead might look like, let's take a look at one possible implementation for an s11n Data Node type, keeping in mind the basic requirements placed on such types by s11n (section 4.2). A basic implementation, not optimized via reference counting, etc., may very well contain the following private data members:

- Two `std::strings`: one to hold the node's name and one to hold its logical class name.
- One `std::list<NodeType*>`, or similar, to hold the children of the node.
- One `std::map<string,string>`, or similar, to hold the key/value pairs of the node. Remember that s11n internally uses lexical casting for POD-type type conversion, so internally all properties are stored as strings. While this might *sound* horrible, this is a simple fact of life and also exists in the world of XML, so i don't feel one bit bad about it. (Besides, most `std::string` implementations are optimized a lot better than most people give them credit for.)

When serializing lots of small objects, this *might be* huge amount of overhead, relatively speaking. i explicitely say "*might be*" because it *really* depends on factors like reference counting, etc., in your STL implementation. As far as i am aware, all STL implementations use such features in their `std::string` classes. Since s11n uses strings *extensively* for storing raw data, s11n can indirectly benefit from such features if your STL provides them. In any case, as the size of the Serializable object goes up, the relative memory overhead of serializing many of them drops. This is little consolation, i understand.

In addition to the memory cost of strings, there is the runtime cost of lexical casting. For string-typed properties a lexical cast is a no-op[51], but properties are often not *natively* stored as strings. e.g. in MyObject, we might store the `change_time` property as a `long int`, and de/serializing that property will cause a short detour through an ostream operation (for serialization) or istream op (for deserialization).

To be clear about all of this "massive overhead", though, consider the following client-side call:

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

Before that function is called, and after it returns, the notorious "second copy" does not exist in memory: it only exists for the life of the serialize operation, and it is thrown away like a used tissue before that operation returns. That is: the cost is an s11n-internal one, and of no *direct* interest to the user, but the user should be aware that serialization *will* eat up memory proportional to the size of the objects being de/serialized (what *exactly* that proportion is, is probably unknowable for all practical purposes).

Remember, too, that client-side objects often also have internal data which is *not* serialized, so the idea that a serialized copy is heavier than the original object certainly does not apply in all cases (mainly it applies to small types - those with only a few POD data members or one container).

[51] In API terms s11n doesn't know the difference between `string` and `int` and `AlaskanPolarBear::MatingInfo`, but some internal optimizing is done to ensure that strings go through as little translation as possible. All that happens, in a worst case, is a `std::string` copy, which is known to be reference-counted in most (all?) STL implementations.

Deserialization *normally* has similar costs: we must build up a tree of nodes and populate an object with the data (creating the object if needed). Where there might be a big difference is the specific i/o handler: if it buffers all of its input/output before it begins de/serialization then the memory costs jumps, theoretically/abstractly by approximately another factor of roughly 1x. That is, it is potentially possible that a de/serialization results in effectively *3x* the memory of an object (again, *very roughly* guestimated). In practice this 3x explosion should be extremely rare or non-existant because:

1. All of the shipped serializers do no special input buffering: they read input stream-wise, creating nodes as they go, until EOF *or* they load one complete root node. This is "buffering" in the sense that we transform the stream content to s11n nodes before passing it back into the framework for deserialization proper, but we do not keep the stream content: it is discarded directly after consumption. There are cases, e.g. networking, where buffering a whole object tree in a string might be required or might otherwise greatly simplify other code.
2. In deserialization we either have an object to deserialize directly into or we have to create one. In either case we have the same as with serialization: effectively two copies of the object's data. The only difference is that in the dynamic-load case we first build up the node tree and then the object, which is of course the opposite of serialization.

It would be interesting to explore a "destructive" i/o API, in which:

- During serialization, we destroy each node directly after sending it down the i/o pipe.
- During deserialization, we destroy the node directly after we deserialize its contents.

These operations are not possible with the current API, due primarily to the required constness of various data. Such operations might also require either new de/ser algorithms or new conventions to accomodate, e.g. a post de/ser functor which algos are required to call on each node. In any case, at some point during serialization we would have a full second copy, but only for a fraction of the time (while de/serializing the deepest leaves of the object tree, since we must dive in depth-first). If i/o support were added directly to a Data Node type and we add such a "destructive" API, then it might be possible to completely eliminate all second copies, at least at the tree level (we might need copies of individual objects). Such support, however, is considered project-specific, and well outside the bounds of the core s11n API. That said, the general s11n model might support such an option, perhaps with a little hacking.

20.5 Runtime speed: s11n and the "Big O Notation"

Please keep in mind that it is architecturally not possible/practical/feasible to impose maximum runtime requirements on the s11n API. For example, we cannot impose the blanket rule that all serialization algorithms must perform their duties in (say) linear time. Stream i/o is one of the places where we simply won't be able to get around paying *at least* linear runtime costs.

As a general rule, most de/serialization algorithms inherently have effectively linear complexity with some constant overhead, but as they may call arbitrary de/serialization algorithms in the course of recursive serialization, they can make no guarantees in this regard. One known exception to the "linear guideline" is the Serializers which do entity translation on their property data (most do this to some degree). The "generic" entity translation algorithm use by s11n is known to perform slowly. i can't name an O notation for it, but it's not a pretty one in any case. i would be extremely happy if someone would contribute a better reimplementation of `strtool::translate_entities()` :).

i will openly admit to having never comprehensively benchmarked nor profiled libs11n. i have run some small speed tests on my standard 1.4GHz PC, and the numbers were well within what i personally consider to be reasonable. For example, an average load-from-stream rate of 20k-50k object nodes per second, depending on the Serializer, and saving is normally faster. Paul Balomiri, an Austrian s11n user, reports using s11n for some 10 million data nodes, 1 gig of XML data, taking 3 minutes to load: this works out to 55k/second, which is close to my numbers.

In my opinion, the fact that Paul can get 10 million data nodes in memory at once without thrashing his system to death really says something about his STL implementation, considering the theoretical memory cost of each node (as explained above). i ashamedly admit that i was shocked and happily surprised at finding out that s11n survived Paul's data set.

i personally use s11n in over half-a-dozen projects, none of which have *nearly* the data requirements of Paul's project. i typically save lists and maps, often nested 3 or 4 levels deep, and very rarely more than 10-20k objects (and normally less than a few hundred). Again, i haven't benchmarked save/load times, but "to my eyes" s11n appears to be fast enough to suit the vast majority of client needs. That is, probably comparable to most home-rolled serialization solutions. In any case, i cannot say that i have ever felt that the load/save times are "too long" - they seem well with reason to me, from a user's point of view.

That said...

There are ways to help speed up s11n if you are willing to look into options like using a customized Data Node type or implementing your own Serializer interface (or subclass). The core library is quite small and 99.9% template code, so it may benefit from compiler optimizations, and "probably" wouldn't directly benefit from most speed-related tweaking. The internals of a Data Node could be implemented more efficiently if one is familiar with that level of optimization (i'm not, really), and the i/o-related code could certainly benefit from some optimization as well. Keep in mind that s11n's core does not rely on the `s11n::io` code in any way, but that s11nlite does. This means that you can use the provided core and your own i/o interfaces if you like. Users who think that such i/o or node type customizations might be interesting options to explore should feel free to get in touch with us through the development list.

20.6 Code maintenance costs

"Code maintenance", in this context, essentially means, "how much time one must write s11n-related code." All software has maintenance costs, and these costs are not always trivial.

It is my firm belief that making s11n any less costly, in terms of maintenance, would be extremely difficult to achieve. In the half-dozen or more projects i currently use s11n in, the s11n-related code is effectively write-and-forget. Once an object is Serializable, it's always a a Serializable, and is usable in all s11n contexts using the same APIs as all other Serializables. Thus once that code is in place and known to work, it normally becomes a pure background detail.

20.7 Money

It would be naive of me to say that deploying s11n is free of monetary costs. As the old saying goes, "time is money", and thus the general rule is:

> s11n's monetary cost of deployment is equal to *your* hourly cost of software development.

That is, every minute of your time it takes you to deploy s11n costs you (or your clients, or someone) one minute of their time. Whether or not that time actually costs anyone money or not is not the point - the point is that deploying anything costs someone some amount of their own personal time slice. (Now if i only had 10 cents for every hour i've spent working on s11n...)

The time-is-money equation is of course nothing new, and applies to *any* software deployed *anywhere*. But we're not here to discuss any software, are we?

i personally consider s11n to have a lower-than-average deployment cost than even most Open Source libraries. The main reason is touched on in the previous section: most client-side code is write-and-forget, rather than write-and-maintain. This means, for example, that implementing a serialization algorithm for a given type (or family of types) is a one-time effort. The exact time it takes to write such an algorithm depends on the complexity of the problem, of course, but by taking advantage of existing algorithms for commonly-understood structures, like the STL containers, we can cut coding times even further. For example, proxying *and* saving a std::map<int,std::string> equates to approximately the following pseudo code:

- `#include <s11n.net/s11n/s11nlite.hpp>`
- 2x `#include` for choice of proxy for `int` and `string` (POD or 1st-class object).
- `#include <s11n.net/s11n/proxy/std_map.hpp>`
- s11nlite::save(mymap, std::cout)

So, the overall money cost can be answered with this question: how long does it take you to do those steps?

As far as the effort it takes to make the average class Serializable - i normally need 5-15 minutes to include all the proper headers, register any proxies i need, write the code, and test it to my satisfaction. Registering proxies for well-understood types - e.g. the standard containers (again) - is a job of under 1 minute, even when typed by hand from scratch. Again, once these registrations are in place, they are background details which needn't worry anyone anymore. Granted, i know the library intricately, but from my client code i behave as client code should (that is, exactly what documentation says to do), and thus in principal any experienced coder can churn out s11n algorithms quickly, and therefor cheaply, once they have done it a few times.

21 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...

21.1 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 list<..>::impl_class()..." or "list<..>::children()." The first hint that the args are swapped is that it's trying to call a `node_traits` function on your Serializable.
- You've tried to pass a pointer as a node argument. *Serializables* are 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 Serializables get some extra leeway (besides, it was trivial to implement the pointer-to-reference translation in SAM). 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 hides that abstraction. 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. s11nlite duplicates/forwards lots of functions simply to keep a whole usable client-side API in that namespace. Be sure to check for differences 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 5. 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 `#define S11N_DESERIALIZE_FUNCTOR ...` to the registration call. This should rarely, if ever, 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. The important thing is to remain calm and simply try to understand what the compiler is telling you. Often a single STL usage error can lead to literaly *tens of kilobytes* of error text (i was once punished with 70k for making a one-letter typo), but after eliminating the first error the others are likely to go away. Elimination of the problem is normally straightforward, once the STL-speak is decoded.

21.2 Containers serialize, but fail to deserialize

See also section 18.2.

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 serialize, e.g. both a list<int> and map<string,string> into the same node, but the result of doing so is undefined. That is, it will serialize, but deserialization may or may not work (don't count on it!).

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 manually. The feasibility for hand-editing depends on the Serializer used: some are not hand-editable. Tips: `s11nconvert` (section 16.1) can convert it to other formats and `s11nbrowser`'s cut/paste features might be useful here (section 16.2).
- 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, or names or nondeterministic) you'll need to do it based on `node_traits<>::class_name()`, 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 its 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 - but such compatibility checks are possible using s11n's proxying model).

21.3 Abstract Interface Types for Serializables

s11n's classloader can handle abstract Interface Types: simply add this line before including the registration code:

```
#define S11N_ABSTRACT_BASE
```

That's all. This does not have to be added for subclasses of that type.

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 an abstract type simply causes 0 to be returned.

22 Evangelism

Obviously, i've got a lot to say about s11n. i mean, how many other Open Source projects of this size have complete API docs, a web site full of example code, *and* a manual of this size ;).

So far i've tried to keep the hype down, but it's sometimes difficult :). In this section i will let loose and explain, in no particular order, some of the library's features which i find particularly interesting, useful, or just downright cool.

22.1 Pointer/reference transparency for Serializables in the core API

That is, the following are equivalent, assuming `list` is a pointer type:

```
s11n::serialize( mynode, list );
s11n::serialize( mynode, *list );
```

One s11n contributor, martin krafft, is always trying to talk me out of this, but the fact is, that subtle feature allows some really amazing code reduction benefits elsewhere. For example, consider what we would have to do for proxies if they had to expect either a pointer or a reference to a Serializable? You got it: we'd have to duplicate every serialization operator for every serialization proxy. No chance i'm gonna tolerate that, so the pointer/reference transparency stays. It is implemented, by the way, via a single template specialization for SAM (a few lines of code). The reality is that these few lines of code *greatly* reduce maintenance costs elsewhere. See the map/list algos, all of which handle pointer and value types with the same code, for some examples of what this allows us to do. Or just read on to the next section, where we evangelize just exactly this technique...

22.2 Container-based algos which are pointer/reference-neutral

Consider these two data types:

```
typedef list<string> StringList;
typedef list<string *> PStringList;
```

i banged by head for quite some time to try to figure out how to do de/ser those via one algorithm. That's not as straightforward as it sounds because for deserialization we need to dynamicaly load the pointer types, and do so polymorphically when possible. Type-dependent branching isn't always *syntactically* possible in C++, so the proverbial *another layer of indirection* was needed to solve the problem of "unified code" for pointers and references. Since the CL layer did the dynamic loading, i wrote up some templates to hide the syntactic and de/allocation differences between pointer and reference types, sticking the CL part behind the pointer-based branch and essentially doing nothing in the reference branch[52].

After some effort and experimentation, a single pair of remarkably small algorithms evolved, and they now take care of de/serializing any standard list, vector, and multi/set. That is, the following operations all go through the exact same few lines of code to do their work:

```
StringList * slist = new StringList;
PStringList * plist = new PStringList;
// ... populate lists...
s11nlite::save( slist, std::cout );
s11nlite::save( plist, std::cout );
s11nlite::save( *slist, std::cout );
s11nlite::save( *plist, std::cout );
```

That demonstrates two separate s11n features: core API transparency for pointers/refs to `slist` and `plist`, as covered above, and algorithm-level pointer/ref transparency for the (`string`) and (`string*`) elements of the lists. The function `s11n::list::serialize_list()` currently does *all* list-based serialization for the framework (that's a LOT). Likewise, `s11n::list::deserialize_list()` does *all* of the deserialization. (Reminder, that's the *default* implementation, and it can be replaced for any specific container type.)

Not impressed, eh? Let's look only at lines of implementation vs. functional scope:

- `serialize_list()` is implemented in approximately 11 lines of non-debug code.
- `deserialize_list()` has approximately 20.

Now consider type L, which is any type conforming to the most basic std::list conventions (this also covers `vector`, `deque`, `set` and `multiset`). Now consider the type ST, which may be any Serializable Type, *including L*. With the above algos we may generically de/serializer any combination of:

```
L<ST>
L<ST*>
L<L<ST> >
L<L<ST*> >
L<L<ST*> *>
L2<L<L3<L4<ST*> > >
```

ad infinitum...

Get the point?

Now consider that we can do the same, using exactly two algorithms, for any combination of standard map-style types (out of the box that's `std::map` and `multimap`, but client-side map-likes can also work with these algos). Let's assume M is a map[SK,SV], where SK and SV are both Serializable types. Now let's begin to look at that more closely, mixed with the Serializabe list type (L) from the above examples:

[52] That "nothing" turned into a long-standing bug-in-waiting, reported by Patrick Lin, which was fixed by adding a one-line "something" in 0.9.17.

```
M<SK,L<SV> >
M<SK,SV>
M<SK *,SV *>
M<L<SV>,L<M<SK*,SV> > >
```

ad infinitum, ad nauseum...
and *Amen, brothers!*[53]
By including the proper proxies, client code gets immediate access to all of the above combinations, plus the *trillions* more they imply. Clients *do* pay compile- and link-time costs, plus fatter binaries, to be sure, but the ease-of-use and coder-effort benefits are, in my opinion, difficult to improve upon. Hopefully, future compilers or development techniques will allow us to cut the compile-side costs. And if not... we'll just need faster PCs ;).
Please note that i'm *not* touting the cleverness of the algorithms themselves, but the flexibility of the s11n architecture, which allows such generic algorithms to plug right in.
If the dimensions of the possibilities don't seem *cool* to you, then s11n probably can't impress you at all (which is all fine and good, i mean - to each his own opinion). However, since this is the Evangelism chapter, i'll go ahead and say: it is my firm belief that s11n supports, *out of the box*, more combinations of data types than most serialization frameworks *could ever hope* to be able to support *at all* (and even then only with unrealistic amounts of client-side or support code). The main reason for this is that s11n takes blatant advantage of newer C++ features which many mainstream libraries shy away from (for compiler portability reasons). My take on compiler portability is simply this: if we want to save 21st-century data types effectively and flexibly, we need to start using 21st-century tools and methodologies.

22.3 "Casting" between "similar" types

Due largely to the above-mentioned features of pointer/reference transparency, s11n allows us to convert to and from "similar" types with ease (though not necessarily with great efficiency). Witness:

```
list<SomeT *> dlist; // SomeT is any Serializable
vector<SomeT> ivec;
// ... populate ivec ...
assert( s11n::s11n_cast( ivec, dlist ) );
```

If the assertion succeeds, `dlist` contains a list or pointers to `SomeT`, copied from the objects in `ivec`. They could be `int`, `char`, `MyType` or whatever - any Serializable will do.
A generic implementation of `s11n_cast()` can be achieved in these few operations:

1. Create a temporary node.
2. Serialize the source Serializable into the temp node. On error return false.
3. Deserialize the node into the destination Deserializable and return result.

The actual implementation looks like:

```
template <typename NodeType, typename Type1, typename Type2>
bool s11n_cast( const Type1 & t1, Type2 & t2 ) {
    NodeType n;
    return serialize<NodeType,Type1>( n, t1 )
        && deserialize<NodeType,Type2>( n, t2 );
}
```

Again, i'm not saying this is a particularly *efficient* way to convert objects, but it is extremely generic. In theory it will work with *any* two types which use the same (or compatible) de/serialization algorithms. Out of the box, that's already millions of combinations, only counting STL-standard containers and PODs (that said, many non-STL containers work flawlessly with the STL-intented algos, as long as they follow the general published conventions).

[53]What would the *Evangelism* section be without an *Amen* now and again?

23 Comparing s11n and Boost::serialization

This section tries to give an overview of the major similarities and differences between s11n and the only other serialization framework for C++ which can provide the range of the features s11n does: Robert Ramey's Boost serialization library, a member library of the Boost.org project. Below we will specifically address points and features which appear in either of s11n or Boost, but probably not in other libraries. Though "Boost" really refers to both an organization and the software that organization releases, here we will use the term Boost specifically to mean Robert's serialization library, which is part of the main Boost distribution as of version 1.3something.

As a software library *user*, if i didn't have s11n, Robert's library would *definitely* be my choice for serialization support. If you are undecided on serialization libraries take a look at the Boost project, which provides not only serialization, but a huge number of industrial-strength libraries: `http://www.boost.org`

Please keep in mind that this chapter is *not* an attempt to sway you away from using Boost! On a coder level, i fully respect Robert's implementation and the design decisions he has made, and am *not* attempting to show that either library is significantly all-around better than the other. However, s11n has only one "competing" product, as far as i'm concerned, and i thought it might be interesting to compare them here. We will assume that the user is familiar with both s11n and Boost, or at least familiar with some of the main design aspects from both.

To open the comparisons on a positive note: Robert and i appear to agree on a great many design decisions. As his docs currently say about this library:

> "Its has lots of differences - and lots in common with this implementation."

A quick comparison of the APIs would suggest that the projects two even co-developed at some point, though this is not the case[54].

23.1 Cans and cannots

Let's take turns listing a few features one lib has and the other does not, considering only out-of-the-box features which clients can get to by following the respective library manuals:

- Boost supports serialization of reference members in serializable classes, at least partially (the support might be fuller than the examples suggest). s11n does not directly support this, though the general model theoretically allows it.
- s11n supports loading without knowing the input format. Boost requires knowing the stream format and using the appropriate handler type.
- Boost internally tracks serialization of pointers and therefor inherently supports serialization of graphs. s11n requires proxies to do this.
- Using Boost in client code effectively requires a hard dependency on much of the other Boost library, whereas s11n (as of 1.1) has no 3rd-party dependency requirements. Likewise, the boost.org libs provides a whole framework, whereas s11n provides only a serialization layer. (We will not count the STL as a dependency in either case because an STL implementation is required by most modern C++ code.)
- Boost directly supports serializing C-style arrays. s11n's author despises arrays and avoids them like the plague, but the framework theoretically supports them: use either a `for()` loop or a `for_each()` functor. The *nature* of both libraries' support is very different because of the fundamentally different pointer serialization policies.
- Boost provides several desireable features which s11n does not: `std::locale` and wide char/string support, `shared_ptr` support, and strong exception guarantees, to name only a few.
- Likewise, s11n has a few interesting features which Boost does not: it overcomes some of Boost's current DLL-related limitations, supports transparent file de/compression, and more data handlers (3 formats in Boost vs. s11n's 8).

Most of these are relatively small differences or express clearly different design philosophies or even simply show a focus in a particular design *direction*. The overal range of features in both libraries is more or less comparable. i believe that both libraries can be used to implement most, if not almost all, of features of the other with some relatively minor internal changes and the appropriate API wrappers.

[54]Robert, you interested? :)

23.2 Compiler and platform portability

Boost has s11n beat hands-down here. Robert has the major advantages of:

- A *lot* more experience than i with multiple platforms. My only development platform is Linux, with occasional access to a Solaris machine. In any case, my practical experience is limited to the GNU compiler and build tools. (That said, s11n is rigorously restricted to ISO-only C++ features.)
- The *massive* peer review effort which boost.org is so famous for. This should *never* be underestimated.
- His software is built on top of other high-quality Boost software (e.g. Spirit does the file parsing), instead off of hand-rolled support code (e.g. the s11n file handlers are mostly implemented in flex-based parsers, rather unfortunately).
- One of Boost.org's core goals is platform-portable libraries. While i always try to adhere to published standards, and never use platform-specific constructs, i cannot personally test or support even a fraction of the platforms out there.

If your software already uses Boost, you should *strongly* consider using the Boost serialization library instead of s11n. i cannot confidently say that Boost-using code would benefit enough from s11n to justify the additional integration costs, considering that a good alternative solution is already available in Boost. While i do believe that s11n provides more features than Boost out of the box, i also believe that Boost could be made to do most, or even all, of the things s11n does with relatively little work. (i suspect that is a side-effect of their STL-ish architectures.) Even more specifically, i think that with the appropriate wrappers, the s11n and boost APIs could probably be made to effectively mimic one-another, at least where their features allow it, as their models are conceptually very similar and inherently very adaptable to this level of modification.

23.3 Archives vs Data Nodes

Boost uses an abstract "Archive" data store concept, which is fundamentally similar to s11n's Data Node model. The main difference is that s11n separates the Node and i/o formats, where the Archive is a combination of data node and i/o marshaler. From a client level there would appear to be little difference in most cases. s11nlite explicitely abstracts away s11n's node type and i/o format, but i believe a similar wrapper would be trivial to add around the Boost code. Then again, the Boost API is simple enough that a wrapper like s11nlite is not really necessary.

Boost's approach is very similar to the model used by s11n's predecessor, which simply had a set of free functions for saving to or loading from the three different formats we had at the time. While it is straightforward and suitable for many purposes, i fundamentally feel that the only s11n-internal entity which should have to know about a stream's format is the code which reads and writes that specific grammar. Even the user shouldn't have to know what format he's using (admittedly, this is a purely philosophical standpoint, not a scientifically-backed one). Actually, the Archive type does not publish any stream-related APIs, even though they work similarly to streams. This means that they can be implemented to be grammar-neutral by simply adding another layer of indirection behind the existing Archiver interface or implementing your own Archiver which uses, e.g. a database as a back end.

s11n internally uses a factory interface for loading all i/o handlers, regardless of whether they are statically linked in with an application or are truly dynamically loaded via DLLs[55], and encourages users to not give a hoot about what data format they are actually using.

One perhaps-not-immediately-obvious advantage of s11n's approach is that it inherently provides the static approach as well as dynamic loading. That is, if you would like to specify a specific grammar handler there is nothing stopping you from doing so:

```
MyClass myobj;
...
s11nlite::node_type dest;
s11nlite::serialize( dest, myobj );
s11n::io::funxml_serializer ser;
ser.serialize( dest, std::cout );
```

[55]It is technically possible to write a classloader which literally creates the classes as needed, but i have never seen this implemented in C++ (the class creation/compilation overhead would be extreme, i think). It's been demonstrated in PHP, for example: creating database classes on-demand by analysing db table structures, creating class code to mimic them, and eval'ing it.

And the converse for loading. You will need to include the proper serializer header(s), of course. The more generic approach, and one which does not require the headers for each serializer is:

```
std::auto_ptr< s11nlite::serializer_interface >
    ser( s11nlite::create_serializer( "funxml_serializer" ) );
if(! ser.get() ){ ... damn ... }
ser->serialize( dest, std::cout );
```

While Boost does not currently appear to offer such a feature, i believe this is largely because Boost currently lacks a cohesive factory API, and this support could probably be added to Boost with relatively little work.

23.4 Non-intrusivity

Though our approaches are quite different, both libs provide functionally similar non-intrusive (i.e., proxied) serialization support. Robert's approach (via overloaded functions templatized on the Archive type) is certainly more portable to older compilers than s11n's approach (mainly via template specializations). i must admit that i simply never thought of his approach before seeing his code, as s11n's model fit so well with template specializations that function overloads were simply never considered. In theory they can be used in conjunction with s11n's model, and *vice versa*. i cannot currently think of any reason why either approach would be fundamentally more or less powerful than the other, nor do they appear to be mutually exclusive in any way. Function overloads are certainly conceptually simpler, and probably *much* easier for new users to grasp, particularly those who are not well-versed in C++ templates.

23.5 Serialization of pointers

This is one of the points where, again, i admittedly stray far from conventional wisdom. Boost takes a very correct approach and has built-in support for tracking the addresses of serialized pointers, such that each is only serialized once and a graph can be correctly deserialized by the core library without user intervention or special support. Boost also has special support for `boost::shared_ptr<T>`, since that is a core component of the overall boost.org framework.

s11n differs quite radically, taking the "convenient" approach of simply treating serialized pointers as non-pointers. That is, serializing (T) and (T*) are functionaly identically. During deserialization we rely on C++'s strong typing support to put us into a context where we can determine whether we need to deserialize a heap- or stack-based object. For example, deserializing data into a list<T*> will create T objects on the heap, whereas deserializing a list<T> will not. This type of difference is handled transparently by the library. The major cost for this is that it (probably) cannot provide built-in pointer tracking support for doing things like de/serializing graphs.

The separation of the core serialization API and i/o API in s11n make this even more difficult, as we need a data-format-agnostic way of building inter-node pointers, so to say. Again, this is a decision which i feel lies way outside of s11n's scope. For example, i don't want someone who uses s11n-generated XML in a non-s11n application to have to conform to the s11n-imposed conventions for embedding references to other nodes in the XML tree. Why not use a standard like those emerging from the W3C? Because s11n is data format agnostic and therefor doesn't know about *any* grammar standards. See the problem? i refuse to enforce force such a requirement on the base Serializer interface, as i feel it would greatly complicate their implementations. Having to write i/o parsers is bad enough as it is, and having to put that much more work into them doesn't sound like my idea of a fun coding session.

Serialization of graphs and other pointer-related tricks *can be and have been* done in s11n, but the core library provides no special support for them. Quite the opposite, the core goes out of its way to hide the differences of pointers and non-pointers!

23.6 Data Versioning

One fundamental design decision which needed to be made very early on in s11n's development was the issue of how to track versions of data layouts, such that we can tell if we are loading data with a different logical version and abort deserialization if we do[56].

This is another one of those points where i seem to disagree with every respectable programmer in the world. *Strongly* disagree, even. My decision was, and probably always will be:

[56] The answer to the whole argument is actually in that last sentence. Tip: the word *layouts*.

Data versioning support does not belong in this library's core. ***Period.***

Of course, it's not fair to make such a strong blanket statement like that without backing up my case. Before i do, a short disclaimer is in order:

> Libraries which do *not* use a key/value pair model for serializing class data *really do require* a built-in versioning system, and a lack of such support in these libraries would *indeed* be a problem. They write X data members to a stream and expect to be able to read X items from the stream, and need some core-accessible way of providing at least basic verification of that. Fair enough.

For reference purposes, let's call Boost's overal i/o approach the "X/X" (or "positional data") model, as it is inherently limited to the physical ordering of the serialized items. We could also call it the Ordered model, but "order" also has other implications which may or may not apply here. In any case, what distinguishes it from s11n, for our purposes, is that X/X requires data versioning to be built in to the core serialization library, whereas a key-value-pair (KVP) model does not.

My case against including this support in the s11n core boils down to the following:

- Doing so requires imposing "some sort of versioning conventions" on all clients. e.g. use incremental numbers or conventional software version numbers, like 1.2.3. This would have been an arbitrary design decision which s11n's author would have to impose on clients. The fewer such conventions the library imposes, the better.
- Doing so requires s11n to have some idea of what constitutes an incompatible version, potentially including support for version number comparisons to allow operations like "support up to 2 revs back" or "compatibility == the same major and minor numbers, irrespective of patch level" or other such oddness.
- How do we report versioning errors? Using the normal return-false approach or a special approach (e.g. version-related exceptions)? Again: that would be an arbitrary decision which s11n would impose on you. The exception approach doesn't (yet) fit into s11n's core conventions, so it was avoided. (This is continually under reconsideration.)
- My personal experiences has shown versioning to be a significant hinderance, particularly for Open Source projects, which inherently tend to fluctuate a lot more than commercial products do. (Mine do, anyway. ;)
- The KVP model, e.g. as used by XML-based applications, appears to be far more version-flexible than people give it credit for. Data versioning can be implemented within this model at theoretically *any level* of a data tree - from the lowliest integer member to the root-most node of a data tree, and it can be done independently of any data format. There are many different ways to implement this, both intrusive and non-intrusive, and it would not be fair for s11n to impose any specific implementation on you.
- Never in my coding life (let's call that 10+ years, if it makes a difference) have i ever needed data versioning for proper function of my applications. If the user feeds us properly structured data, deserialization works, otherwise it fails. Why make it more complex than that? As in XML-based application, semantic validation is necessarily a client-side choice and versioning falls into the category of semantic validation. s11n concerns itself with the *structure* of the data, and cares very little for the semantics of the data (and then only for classloading, because we have to store a unique-per-type identifier to a C++ class).
- And finally...
  Computers are inherently stupid, and the thought of a piece of *software* telling *ME* what data i am *permitted* to feed *MY* application makes me queasy. It makes me downright mad, actually. This is OUR decision to make, not s11n's, and s11n's architecture allows us to make such determinations at almost any given point in the deserializaton process, should we want to.

A quick, incomplete comparison of the properties of each model reveals the following notable practical differences:

- The X/X approach is grammatically more compact, potentially *drastically* so. For proof of this just compare any XML file to the equivalent in a binary grammar. The addition of client-transparent stream compressors (e.g. built on top of zlib or bz2lib) makes this point largely moot, at least for practical purposes (though not techno-philosophically, because such features are not always readily available in all projects).

- The KVP approach writes *named* elements and can search for them later by name. Thus we can add properties, remove them, check under different names for the same property, and other operations related to version interoperability. That capability is not quite missing the X/X approach, because we can map version numbers to specific deserialization operations, but we don't have the playroom which KVP allows for.
- The X/X approach would appear to require more maintenance than KVP-based code when a class gets new members. Robert's X/X implementation is quite sane, but still requires some amount of care on our part if we wants to support older data files as our objects change, if only because (a) each developer has his own philosophies about version numbers and (b) the version number is defined at one source code point and accounted for at another point, which makes them easy to get out of sync, especially in multi-developer projects. In X/X, a failure to change a class's version number when its serialization algo changes (e.g. as data members are added or removed) can result in unpredictable, or even undefined, runtime behaviour. (i believe Boost explicitely throws if it detects this problem, but i am not 100% certain of that. RTFM.)
- The X/X approach possibly provides easier trackability of pointers when doing things like serializing graphs. Theoretically, though i can't really back that up at the moment. s11n's "deep pointer copy" policy shifts such "special-case" work to the client, whereas Robert's code handles all of this transparently.
- Data files created for X/X models are inherently unusuable by KVP models, but the other way around is not the case because we can always discard name info later to create X/X data from a KVP data set. It is interesting to note that Robert's documentation shows an example of serializing using a KVP interface, in which the key is internally dropped.

Which approach is better, KVP or X/X? As always, it really depends on what your needs are. i obviously prefer the KVP approach, and personally consider details like data compactness to be " issues of the past" (so sue me - i almost always choose convenience over drive space).

23.7 API ease of use

Learning Boost is probably much simpler to get started with than s11n is. Boost's public API very straightforward, even almost intuitive. While s11nlite's public API is just as simple, s11n sets out to specifically abstract away a couple more details than Boost does and has a proportionally (perhaps even disproportionally) higher learning curve. For example, Boost does not appear to have a public factory/classloading layer, so those details never come into play.

Once the learning curve is climbed, s11n and Boost have approximately the same ease-of-use, i think.

Boost also takes advantage of operator overloading to provide a simplified client-side API. For example, if A is some Archive object and S is some serializable object, you can probably guess what the following operations do:

```
A << S;
A >> S;
```

Fundamentally, this shouldn't be a problem to add to s11n. Practically, however, s11n's use of the `node_traits<>` type as an API marshaler for arbitrary node types complicates the matter, as the operators would really need to be part of that `node_traits<>` interface. While i haven't tried it out, i do not believe it would add to s11n's ease of use the same way it does in Boost, mainly due to having to either create a traits object to apply the operators to.

Additionally, s11n's i/o model would inherently complicate such an addition, as discussed in section 18.4.

If a user is willing to stick with a single concrete data node type, such operators could of course be part of that API. i am not keen on the idea of adding them to the core node interface, however, even though in Boost's case i do consider them to be justified.

23.8 Serialization Traits

That s11n and Boost both use traits types to store information about serializable types is pure coincidence. We both use them for tying metadata to types for purposes of managing serialization, but we do *completely* different things with them. Boost manages, for example, pointer tracking, custom RTTI info, and data version number (a very clever place to put it, actually), whereas s11n mainly uses it for providing typedefs and (as of 1.1) access to class names (which is conceptually similar what Robert does with his RTTI info).

It was by reading the Boost documentation that i learned that s11n's proxying and traits approaches will only *properly* work on C++ platforms which *fully/properly* support partial template specialization. On others it might not choose the proper specialized types. i have *no idea* what compilers might be troublesome here. Not mine, anyway ;).

23.9 Efficiency

Again, Boost has s11n beat hands down on this, on all accounts.

One of the main reasons is that Boost uses parsers written using Boost::Spirit, a true wonder of technology which obsoletes tools like lex for C++ projects and generates code which compilers can theoretically optimize down to the last bit. The unfortunate fact is that most of s11n's input handlers are written in lex, and this includes a rather large amount of underlying support code to help lex code fit into the modern C++ world more satisfactorily. This is not something i'm proud of.

i would love to use Spirit in s11n, and have wanted to for over a year, but i always had problems building it on my boxes, and thus never came to depend on it. i hope to include Spirit-powered parsers in s11n someday, because Spirit is just too cool to overlook: `http://spirit.sourceforge.net`

To be clear, neither Boost nor s11n inherently rely on either Spirit or lex, or any other parser framework for that matter, but a serialization library without *some* form of included i/o support is pretty useless for most cases (but not all cases[57]!). This i/o support takes the form of some type of parser, but this is largely an implementation detail and normally need not intrude on clients at all.

Another area where Boost is inherently much faster than s11n is in it's one-pass de/serialization model. The Archive type *is* the i/o marshaler, and all de/serialization operations are performed directly on Archive objects. In s11n we de/serialize objects from/to containers, similar to how we would in an Archive, and it is these containers of "raw" data which are used by the i/o handlers. This is an unfortunate cost of the physical separation of core serialization operations and stream i/o, but one which i believe is highly justified for this library.

That said, it is theoretically possible to add internal i/o support to a new Data Node type and use that node type with s11n to provide similar functionality as Boost's Archive type. Likewise, it is theoretically possible to similarly wrap up Boost's Archive type to use two-phase de/serialization (as if you'd want to). Both architectures are very flexible to this type of change.

23.10 The interesting part is...

In hindsite (after having written this chapter, which included reading much of Robert's documentation and some of his source code), the following points have become clear to me:

- Robert and i are indeed, as he once said in an email, "kindred souls," both out just trying to save our objects.
- On the surface, s11n and Robert's code have a few similarities. All coincidental.
- At the overall architecture level, they have an *uncanny* number of similarities. Again, all coincidental.
- The implementation details are *completely* different animals.

That last point, in particular, strikes me because what's *really interesting* about it is: they are different animals *for completely different reasons.* That is, the features Robert's code and s11n provide are not necessarily mutally exclusive, but often exist either as different approaches to the same end or as solutions to completely different parts of the overall serialization process. In some cases each goes into areas the other simply has not explored. A couple examples include:

- Robert's Archive and s11n's data node models are not only both there to serve the same end (the client's interface to and from The Void), but also are both important templatized types for the architectures.
- Robert's Serialization traits types track pointers, version numbers, and RTTI info, amongst other things. s11n's traits provide the `class_name()` function, which logically overlaps somewhat with the RTTI features, and several functors which play a similar role as function overloading does in Boost's code (and it does so in a mutually compatible way, it turns out).

[57]There are actually valid uses for serialization without any underlying i/o, like databases, shared-memory (where objects could be written directly), and other such "exotic" cases.

The main implication of this would seem to be that it might be completely worthwhile to look at either merging in features from each other's library or to work out some way to merge them. A simple disappearance of one of the libs would not be acceptable by either of us, i'm certain, and i do feel that both distinguish themselves enough that they cannot simply merge one-to-one. It would be interesting to figure out how the core differences of, e.g. versioning and deep vs. shallow pointer copying, could be abstracted into policies or other C++ techniques, such that we could present a single core and build our own features on top of it. After having read much of Robert's documentation, i have little reason to think that this is not possible. The difficult part, i think, is figuring out where the line between core and client-side policies should come in. Something to think about, anyway...

23.11 In closing: s11n.net and Boost.org

To be clear, no s11n.net software has *any association whatsoever* with Boost.org's software, and we won't defame them by claiming any such association.

From here on we switch from "Boost" meaning "Robert Ramey's Boost serialization library" to Boost meaning the Boost.org libraries in general.

Several people have written me to ask if i plan on submitting s11n to Boost.org for consideration as a member library.

i'm *truly flattered* by this question, but i have no plans on submitting s11n to Boost.org. The reasons are:

> (Please accept my appologies in advance if any of the reasons below seem presumtuous, pompous or even downright stupid. Everyone's got their own quirks, and a several of mine are expressed below.)

- They are Gods, i am not[58]. They would eat me alive and call it a Virgin Sacrifice Breakfast. i am a long-time hobby programmer who's hobby serendipitously turned into not only his profession but also his lifestyle. A virtual hippie, so to say. By comparison, many of the Boost members are well-trained, seasoned veterans of far more design committees and software wars than i.
- i believe the Boost team would (quite rightfully) try to enforce a strict set of exception conventions on the library. As discussed in section 4.6 of this manual, i currently have reservations against doing so. i don't want to be faced with that reality quite yet. The day will likely eventually come, but only after i feel comfortable with all of the design decisions and their implications.
- i would likely be required to explicitely support, or help to support, a wide variety of C++ platforms which i will never in my life lay fingers on. i could not, in good conscience, possibly claim to support platforms whos names i know only from `#define`s in `config.hpp`[59]. Likewise, i despise spending a significant amount of my coding time researching workarounds for deficient platforms, even if i *do* have access to those platforms. i *love* software development, and i want it to continue being *play-time* instead of turning into *work-time*.
- My freedom to experiment in the main source tree would be more limited, as stabler interfaces would be that much more important. Either that, or i would end up maintaining two different copies. That would not only be a real drag, but would also send the wrong message to users by providing two potentially incompatible APIs.
- While there are some compelling differences between s11n and Robert's implementation, our libraries are uncannily similar in both nature and design. i believe that in bending s11n more towards Boost we would end up at roughly the same implementation, or at least very similar features wrapped up in very similar interfaces. Neither Boost.org nor its users would benefit from an overlap of that size, even if "some competition within Boost might be a good thing" (as one writer suggested). Boost is a cohesive whole, and non-duplication of features helps keep it that way.
- We could probably never get a group concensus agreeing to keep s11n's deep-pointer-copy policy (more likely, i would be outvoted 400 to 1). Nor would we ever find a 100% all-around-agreeable factory interface, including the underlying conventions. Nor would most efficiency-seekers even look twice at s11n's heavy use of lexical casting, would demand internal native type support via, e.g. `boost::Any`, would require strict performance definitions, etc. Fair enough, but that simply isn't my thing.

[58]"Ray, the next time somebody *asks* you if you're a *god*, say *YES*!" – Ghostbusters

[59]i'll save the *Tirade on the Illusions of Portability as Perceived by Most Autotools Users* for another time.

By and large, i'm worried about Death by Committee even more than the death via Virgin Sacrifice Breakfast, though i'm not sure who would die first, s11n or my desire to continue coding on it.

To be absolutely clear: both this library and i *would* certainly *both* benefit *greatly* from the Boost code review process[60]! Well, the one of us who didn't die first would, anyway ;). i want to save my objects *now*, and s11n does that *now*... and does so *without killing anyone*[61] :).

It is possible, but i don't quite dare say "likely", that i will at some point fork off a copy of s11n which is based off of the core Boost libraries, targeted specifically at Boost-using client code. This primarily depends on the availability of Boost on client machines (traditionally it is not preinstalled on most systems).

One of s11n's long-standing design decisions has been to reduce 3rd-party library dependencies to a minimum. Thus i spent 2+ years writing utility code which already exists in libraries like Boost :/. If we were to replace all of s11n's "utility code" with Boost equivalents, we could probably cut the size of the tree by 1/2, not counting the i/o parts (that makes up the majority of s11n's code). And i could *finally* get rid of that damned string utility library which keeps hopping from source tree to source tree like a little virus.

Assuming even a modest 20% code reduction, that would equate to 20% less code to maintain, which is always a good thing. Of course, it also means relying on gawd-only-knows-how-many underlying libraries in Boost, the interfaces and behaviours of which we can only hope are stable from one version to the next. (To be clear, i have no experience with Boost version compatibility, so i am not badmouthing them here!)

Not to be underestimated: some of the Boost code will theoretically become part the "next" C++ standard library and it would pay notable maintenance dividends to base s11n off of these libraries as much as possible.

i feel compelled to make a final confession, as well, and explain the reason why s11n is not already built off of the Boost libraries. This has been asked more than once, and the question is a fair one.

i have some deeply-seated, admitedly somewhat eccentric, philosophical problems with the Boost distribution policies. Not their licence, but the *way* their code is distributed.

In short, my message to the Boost team is this:

> If the code was easy to install, *i would have been using Boost since years.* Please provide *some* form of conventional build process. Whether or not they are Autotools, *i don't care*: a simple configure script and/or Makefile would do. Justification: as a library coder, if i do not believe that Library ABC will be on my target client systems, i generally will not introduce a dependency on Library ABC in *my* libraries. i'm pedantic about that, to the point of even skipping over jewels like Boost if their value isn't relatively convenient to cash in on.
>
> *And get rid of the `config.hpp` "feature" of `#error`ing on the unknown compiler version every time i upgrade my gcc!!!!! ARGH!!!!*
>
> i admittedly get overly-annoyed when it comes to points like these, but if you guys will fix these things then i'm your newest convert for life. The wonderful code - and even complete documentation - is all there. Practically a C++ Nirvana right before our drooling mouths, but it is nonetheless not as accessible as it should be.

Potential Boost users: please pay no attention whatsoever to this man's ramblings - give Boost a try and you will probably be amazed by its quality and range of features.

24 Is this the end?

We are nearing the end of the document, but hopefully the new possibilities for saving your data have just begun. :)

If you are looking for more information about using s11n, try:

- The s11n source tree has code for a couple client-side apps, which will certainly prove informative to those starting out with s11n.
- 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`

[60]TODO: see if there's a way to Boost-supported process to submit code for review with the explicit idea that it is not targeted at inclusion for Boost. i suspect not, given the necessary overhead, but it would indeed be very interesting. A "Boost of Breed" stamp of approval type of thing.

[61]At least no such bugs are currently known.

- If you have questions, concerns, or just want to say "Hello, world", please email us: `s11n-devel@lists.sourceforge.net`

.....

Before i go, i want to tell you briefly why *i* use s11n in all of *my* code: because it's just so damned easy to do. When there are such time- and feature-gains to be had via such a simple-to-integrate tool, it's hard to justify re-implementing any save/load code[62]. This continual interaction with multiple clients also greatly helps in figuring out exactly what s11n needs to do and what services it must provide, so the library continually reshapes and improves under the well-proven and very-very-very long-standing rules of Natural Selection, also known as Darwinistic Processes or, in the marketing department, Upgrades.

As always:

- The source tree is *always* the most-definitive source of information, but the web site is also updated fairly often as new advances are made, often a bit in advance of upcoming changes.
- i am always open to getting mails with questions about s11n, so don't hesitate to email our development list. 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, and i will be happy to fill those holes by answering your questions.
- The main goal of s11n is to *Save Our Data*! If s11n can't do that, please help us out by suggesting how we might be able to change it so that it *can* save your data! Sometimes just saying "s11n can't do [this]" is enough to spur a solution, as often the author does not realize something is a problem or omission until someone else points it out (thanks again to Ton and Gary, especially, for that).

Once again: thanks *a lot* for taking the time to consider adding s11n to your toolkit! And thanks *a whole lot* for Reading The Full Manual. :)

—– stephan@s11n.net

or, of course:

s5n@s11n.net

:)

Happy hacking!!!

[62] You can bet your emacs that i'm pretty sick of that part by now ;).

Index