Making a Boost Library

Robert Ramey
Software Developer
830 Cathedral Vista Lane
Santa Barbara, CA 93110
(805)569-3793
ramey@rrsd.com

ABSTRACT
Boost is a loose organization of C++ developers dedicated to the creation of high quality C++ libraries. It can be found at www.boost.org [1]. This article describes the process of getting a library accepted into Boost along with advice from one who has been there.

Categories and Subject Descriptors
D.2.13 [**Reusable Software**]:Library Software Development.

General Terms
Design, Human Factors, Legal Aspects.

Keywords
C++, Boost, Libraries.

1. WHAT IS BOOST?
Have you ever wanted to:

- do a really, really good job at something?
- provide the "definitive" or best solution to some problem?
- make something that lots of really smart people would appreciate and use?
- work to a higher standard than your current job requires or will permit?
- demonstrate that you are really a good programmer?

Maybe you want to consider making a library and submitting it to Boost.

www.boost.org is a loose organization of C++ developers dedicated to the creation of high quality C++ libraries.

Boost libraries are distinguished by:

- Wide applicability – libraries are usually things that are widely applicable. The effort required to write a library and get it accepted to Boost is not justified unless the library is going to be re-used many times. For this reason, many (though not all) Boost libraries are fundamental building blocks like smart_ptr,
- As a corollary to the above, Boost libraries are portable. They are written to the C++ language and library standards with work-arounds for bugs in specific compilers. Most Boost libraries leverage on idioms already in boost which already have been implemented in a portable way.
- Another corollary to the above is that Boost libraries tend to strive to be the "best" or "definitive" solution to a particular problem.
- There is relatively little repetition of functionality within Boost. If there is a best and/or definitive solution to a problem, other libraries generally incorporate it.
- Boost libraries often use cutting edge techniques such as template meta-programming to achieve desired goals.
- Boost libraries strive for high quality. This is attained via an exhaustive testing discipline and corresponding infrastructure.

The above common library features are the result of a very public, rigorous and iterative peer review process that draws on the experience and knowledge of the entire Boost community.

Boost libraries cover a wide range of functions and applications. Among the most wildly used are regular expression parsing (regex), smart pointers (smart_ptr), threading, date time, file system, preprocessor, testing and correctness and others. It is really not possible to convey in a short paragraph the breadth of these libraries. A complete list can be found at [4]

All Boost libraries are subject to the Boost License [2] which is designed to permit usage of the library as widely as possible.

As the author of the recently accepted Boost Serialization Library, I can attest that making a library and getting it accepted into Boost is much harder than it would first appear. This article describes Boost and what it takes to get a library accepted. Note that opinions and advice expressed here are my own. I do not presume to speak for any other Boost members.

2. THINKING ABOUT YOUR LIBRARY
We all have at least a few really great ideas.

2.1 Some Ideas Are Really, Really Hard to Implement.
Some things are inherently difficult. One recurring idea is a dimensional analysis and units library. C++ operator overloading makes this idea very appealing, so it is easy to get started. There are many libraries available in this domain and several have been submitted to Boost. None has yet reached the formal review stage. This may be because there is wide applicability of such a package and it is very hard to reach a consensus on requirements and implementation. Many people need dimensional analysis and

have made and used libraries that suit their needs. Making one library that covers enough applications may be just too difficult, so no consensus has been reached.

2.2 Consider Making a Smaller Library

As we will see below, making a Boost library and getting it accepted can be a huge undertaking. Before embarking on the process, you should consider if you can see it through. It may be a better choice to make a smaller library.

Many of the most useful and widely used libraries, such as STATIC_ASSERT, are small but tricky.

Even a small library will entail more work that you might think. Better to make something small that is really useful rather than something bigger that does not get finished.

2.3 Start Writing Documentation.

I know that seems backward to a lot of people, but bear with me.

- Description – what this library does.
- Motivation – why is such a library useful?
- List of features required by such a library.
- List of other libraries that do something similar and how your library is different and or better.
- To get started, you will have do some research. Be sure to include the Boost website in your search:
- Website. Something like your library may already be in Boost. Or perhaps something you can build on is already there.
- Mailing list. Here is all the information about previous proposals and submissions. It is quite possible that something similar to your library has been submitted in the past and not been accepted for some reason. If so, you need to know it. It is also possible that the problem your library is intended to solve has been discussed.
- Files section contains libraries that have not been formally accepted into Boost, for various reasons. Some may be in process of development. In many cases the library is more of an experiment than a full blown library. The author might have submitted the library but did not have the time to push it all the way through the process. In my view the files section is an underappreciated gold mine of useful code. I look through it all the time when I have a small sticky problem. Need to render in integer in roman numerals? It is in there!

Your library should leverage facilities already in Boost rather than re-invent any wheels so you can spend all your time concentrating on the unique aspects of your package.

At this point, Boost recommends that you query the list to see if there would be interest in your submission. This is commonly done. Personally I do not think such a query is always a great idea. If you have done your research, you should have a good idea whether or not your proposed library will be interesting. When you query the list you risk getting involved in an opinionated discussion that revolves around a still nebulous idea. My view is that the real issues do not present themselves until some code is written, tested and compiled. My (silent) reaction to such queries is: Hmmm – might be interesting, let's see the code and some documentation.

2.4 Become Familiar with Boost Tools

Start out by installing the current Boost libraries on your development system. Boosters think this is easy. And it is, after you are familiar with it. It means getting paths and environmental variables setup for the command line version of your favorite compiler and a couple of other things. Unless things go perfectly the first time, you will have to investigate how the build system works which takes some time. For this reason lots of users of Boost libraries just incorporate Boost source code headers into their projects. Many of the Boost libraries are supplied as header files and do not require the building of linking libraries. However, for a library developer you will have to become familiar with the whole system.

2.4.1 bjam (boost jam)

Boost has its own system from building executables and running tests. It might best be described as a next generation of UNIX make. The main component **bjam** processes a Jamfile which describes the requirements for buiding libraries, and executables. Dependencies between header and source files are handled automatically. Also compiler, library, and platform dependencies are also handled automatically. Generally, there is little or no compiler, library, or platform specific information in a Jamfile. In this way, your library will be built and tested on other platforms without anyone having to do anything special. Of course that is the theory. In practice there is usually a little bit of effort required to specify small adjustments required for different environments Without **bjam**, it would be a huge effort just to test someone else's code – now it is manageable.

bjam is used to build libraries and also run a test suite for each library. Information on using bjam is spread among several web pages on the boost site. Perhaps the easiest way to get familiar with bjam is to use the bjam files for other libraries as models for your own.

Unfortunately, it is one more thing to learn and at the beginning it will feel like its slowing you down. In fact, it IS slowing your down. But the investment in effort to become familiar with it will be paid back many fold as your library becomes more elaborate and ported to more platforms.

2.4.2 Documentation

As I write this, most of boost documentation is in HTML files. This is considered acceptable for new submissions. These files may be generated by hand or with another tool of your own choice. To save time I used a skeletal set of HTML files from boost that provided all the sections, and style information that is common to boost libraries. I found this very helpful. Boost is moving towards a new system for documentation, Boost.Book**,** which maybe worth investigating.

2.4.3 Boost Test

Fundamental to a library submission is a test suite. The key tool for building tests is the Boost Test library. This described in the Boost library documentation in the section "Correctness and Testing" [5]. Tests are run on separate test servers and produce a daily test matrix which shows all test failures organized by library and compiler.

2.4.4 Other Boost Tools

It is really necessary to have reviewed most of boost libraries to understand what is available already. Boost contains lots of code to simplify program portability, ensure correctness and implement commonly required idioms. Code that needlessly includes functionality already in boost will probably not be accepted. It takes a little time to become familiar with all this.

3. CRAFTING YOUR SUBMISSION

Now you are ready to write your code. More likely, you have got your original code and you are ready to start making it acceptable for Boost.

In order for your library to be evaluated, others will have to experiment with it, test it and use it.

If someone wants you to try out their code, they had better make it as easy as possible for you – correct? You could not spend time fiddling around with compiler settings, deciphering incomplete, or unclear documentation or otherwise wasting time. Well, surprise, no one else can either. Be prepared to submit a self contained package that "just works". Given that the B**oost** community uses a variety of compilers, libraries and platforms, this challenge might seem impossible at first glance. Boost tools provide a solution to this problem, so now you will start modification of your code to do it the "Boost way".

Code, make test, add to Jamfile, debug, add to document, redesign, re-factor and repeat until done. Soon you should have the following in your personal copy of the Boost directory tree:

- Code for your library's headers and source files.
- Code for tests and demos
- Jamfiles for build and test
- Documentation for library usage

When crafting your library:

- Work to the C++ standard – not to a particular compiler.
- When the compilers you use to test cannot handle the standard conforming code, make changes to achieve portability desired. Use facilities already in boost to achieve the desired portability.
- Use at several different compilers. This will increase the number of people that can/will tryout your library. All compilers have bugs and quirks. Building your code with more than one compiler/library helps make your code more portable and standard conforming.
- Leverage other Boost libraries to achieve portability and gain "free" functionality.
- Add a section to your documentation titled "Rationale". As writing on your library progresses, you will be required to make non-obvious design and implementation decisions. For each of these decisions, add an explanation to the "Rationale". Later, when it is asked why you did something a certain way, you will not waste a lot of time re-discovering your original reasons.
- Include a tutorial with an example program in your documentation. This should permit an interested party to see the utility and ease of use of the library in a very short time. In a sense, the function of this section is to "sell" the library to a potential user..
- Include reference documentation to catalogue its features and usage.
- For each library feature, include a test and add it your Jamfile. Also add an entry to your reference documentation.
- Repeat the cycle, adding features until the library is mature enough to demonstrate its utility, design, direction and final form. It does not have to be complete, but it should have functionality that others can benefit from, including working code, documentation and tests.

Eventually you should have enough of the library done that it can be evaluated. The documentation and code might have some placeholders but it will be clear what you envision as the final version. Now you are ready to submit to Boost for preliminary consideration. Announce your submission on the Boost developer's list and make it available to interested parties in one or more of the following ways:

- zip, gzip, or tar your submission into one file. Be sure to retain the Boost compatible directory structure. This can be uploaded to he Boost files section and/or to your personal website.
- Request CVS access to the boost-sandbox project and check it in there.

If all goes well, you should get some feedback on the list within a couple of days. If you do not get any feedback, try announcing again as sometimes the list is focused on a heated technical debate, formal review in progress, new release or something like that. The Boost community is large so usually there is someone interested in just about any topic posted.

Hopefully some people will find your package interesting, useful and transparent enough to experiment with.

4. DEALING WITH FEEDBACK

Now the fun begins. Hopefully you will get some feedback. Hopefully at least some of it will be positive. Here is what you might get.

4.1 It's Got Bugs

Well, shame on you. You did not test it exhaustively enough – this is a big turn off for users. Think of your own reaction when you have a problem, find something that purports to solve it, and it turns out to be more trouble than its worth. You are disappointed and reluctant to trust the library (and its author) again. Do not do this to your users or to yourself. Better to have something useful that has a path to the future than something that does not work along with a promise about how great it is going to be.

Maybe it is a misunderstanding – ask the user to run your tests – or add a new test.

4.2 It's Useful But It Needs Feature X

Now we are getting somewhere. Someone is actually trying to use it. This is a big accomplishment. Feature requests come in lots of flavors.

- There is a way to do it – it is just not clear from the documentation and examples. So it is just a misunderstanding. This is your cue to add another section to your documentation with a supporting example and test.
- It is already planned for the future. Very good – they want more. Acknowledge the request and put it your list.
- You never considered it, but it is a good idea. Great, put it on your list.
- You are convinced feature X is not a good idea. – explain why this feature is not included and not planned. This may start a discussion thread. If you already considered this and rejected it, it should already be in the "Rationale" section of your documentation. Eventually it will get sorted out and hopefully a consensus will be reached. Be sure that the entry in the Rationale reflects the relevant aspects of the discussion.

4.3 How Do I Use the Library To Do X?

This question is similar to a feature request. It will result in either a promise to add a new capability, an example or demo with an attached explanation showing that the capability is there, or a new entry in the documentation indicating why the capability is not there.

It is often easier to write a small demo showing how to do something than it is to explain how to do it. It is much better than a seemingly endless back and forth on the mailing list. Add the demo and explanation to your test set and documentation. If you do not do this, the same question will come up again and again.

4.4 Personal Comments

Boost mailing list discussions are governed by the Boost Discussion Policy [3]. This policy is designed to make mailing list discussion as productive as possible and avoid problems which can plague other mailing lists. Among other things it proscribes personal attacks and admonishes list members to stay on the subject. Members on the list use their real names. Occasional gentle reminders keep this list functioning in a product and professional way.

Of course someone might slip and say "This feature is useless" rather than "When would I use this feature?" which is far more likely to elicit a useful response. Should something like this happen do not take it personally. Often what seems offensive is a communication problem. Remember that the Boost list is a world wide community. Sometimes there can be misunderstanding because of language difficulties. Unless you are in a position to respond to a poster in his native language, have some patience. Generally I like a little humor in posts – but remember that it can be the source of a misunderstanding and someone can be offended when there was no such intention.

Often the poster may have a valid point even though he phrases it in an irksome manner. Just respond to the substantive point and ignore the rest.

When arguing issues related to your library, stick to the Boost discussion policy.

The most heated discussions revolve around differing opinions and hypothetical situations. Often time the best thing is to make a small test program and get some real facts. That will usually resolve things. In any event, the discussion will move to a higher plane that revolves around interpretation or applicability of real results of a real case rather than speculation based on hypothetical situations.

4.5 The Next Step

Well, you have your input. Most likely you have a long list of things to do. Some people find the feedback discouraging; other people find it motivating. If you are convinced that your library is on the right track, be prepared to repeat the above procedure several times. The last draft of the Serialization library is # 20. About half of these revisions were posted to Boost and actually subjected to the process described above. The serialization library is larger than most and I really was not prepared for this process the way you will be after reading this paper. Hopefully, my experience qualifies as a worst case scenario.

5. FORMAL REVIEW

Formal review is the heart of the Boost process. It is fiendishly clever and very effective. It can be summarized as follows:

- Formal review is requested by submitter
- If the request is seconded by one or more Boosters, a limited time review period (usually a week or two) is scheduled and a review manager is assigned.
- Issues such as library design, utility, code quality, documentation, and others are discussed on the list.
- During the review period, interested parties post recommendations and supporting arguments for acceptance or rejection of the library into Boost.
- After close of the review period, the review manager makes the decision as to whether or not the library will be accepted, and if so, what changes should be made. His report includes a summary of the issues raised and his assessment of the consensus.

A particularly intriguing aspect of this process (to me) was the lack of pretense to any sort of democratic idea. Although reviews often "Vote" for acceptance or rejection, it is not a question of number of votes. The review manager makes the final decision after reviewing all the posted comments. It is much more akin to a court decision rather than an election.

The fact that this is a formal review will motivate a number of people who did not have time to review the library before to now take a closer look. Having updated your library in accordance with your preliminary feedback will pay big dividends here. Less time will be spent on mostly settled issues so you will be able to spend time on any new things that pop up.

The formal review process itself can be pretty intense for the library submitter. The limited time frame available focuses

everyone's attention on the review. Many new points will be brought up and you will have to consider them all in a short time. The process sounds more suspenseful than it really is. By the time this is done, it is usually obvious whether or not the submission will be accepted.

If your library is not accepted, the review manager's report will detail the reasons why along with the final decision "The X library is not accepted into Boost at this time". Of course such a decision is a huge disappointment for the submitter. Though it has happened that a library which was deemed unacceptable was reviewed a second time, it has happened only once. The Boost Serialization library is the holder of that dubious distinction. A better strategy is to be ready the first time by following the advice given here.

6. SO YOU THINK YOU'RE DONE?

If your library is accepted, it is usually subject to some conditions. Boost does not require the library to be totally complete to be accepted. Accepting only libraries that are ready for release would place an unreasonable burden upon potential contributors. Your next task is to make all changes that the review manager has determined are necessary. This can take quite a while.

Once all the changes are made, you can concentrate on other portability to other platforms. Boost emphasizes that support for older non-conforming compilers is not a requirement. Whether you choose to implement conformance workarounds may depend on the nature of your library. If your library is the next greatest template metaprogamming wizardry, it may not make sense to try to support older compilers. If it is a more prosaic application such as a TCP/IP stack, it might be more appropriate to support a wider range of compilers. It is up to you.

Buiding and testing with other compilers, libraries and platforms can be more difficult that one might think. First of all, if you have made it this far, your library may have lot more functionality and generality than it started with. You will start to gain a better appreciation for the subtleties of the C++ language and the variations among implementations of the language. Eventually you will add your code to the main Boost CVS tree and start testing on other platforms. Boost runs all the tests for all the libraries approximately every 24 hours. The slow turn-around can be infuriating. Fortunately, friendly Booster members interested in your library will often lend a hand with the compilers, libraries and platforms that they use.

Eventually, most of the boxes in the test/compiler test matrix show the tests passing. A few will not pass because one or more compilers or standard libraries cannot support a particular feature that your library requires. (e.g., wide character I/O). Some compiler bugs just cannot be worked around, so some feature of your library may not be usable with a particular compiler. This matrix will help library users to determine which features are available in their development environment.

Is this the end? Not really. Libraries are constantly tested and new problems emerge as compilers are upgraded. Users report ever more bugs or ambiguities in documentation. Users post suggestions for enhancements. Depending on the size of the library and how widely used it is, it can take a while before things really taper off.

7. IS IT REALLY WORTH IT?

Submitting a library to Boost and seeing it through can take a lot of time. It can be frustrating and stressful as well. And for all this, there is a real possibility that the effort will end in failure. The question has to be asked – is it worth the effort?

Regardless of whether or not your library is accepted, you will benefit from having gone through it.

You will find that there is a lot more to C++ than you thought there was. As a library writer you will likely become a lot more familiar with the details of templates, STL, streams, etc. than you do as an applications developer.

You will be exposed to better methodology. The Zen of Boost might be summarized as

- Design, code, tests, and documentation are developed in parallel rather than one after the other.
- Development is incremental and iterative. During the course of development, one always has a complete working package.
- Subject code, tests, and documentation to constant review and criticism of one's peers
- Factor out common code into libraries of orthogonal functionality.
- Test each library and each library feature independently.
- Document libraries separately.
- Composing programs from working, tested, documented components increases the chances of producing flexible, reliable programs in the shortest time.

Most organizations believe that they are using the best practices to produce software. Most of these organizations are wrong. Going through this process – even for a small library – will make it apparent what it takes to do good work and why more of it is not being done.

You will spend time interacting with smart, mostly agreeable people who really love what you – and they – do.

Is this "worth it"? You decide.

8. ACKNOWLEDGEMENTS

David Abrahams and other Boost members critiqued this paper.

9. REFERENCES

[1] www.boost.org

[2] http://www.boost.org/LICENSE_1_0.txt

[3] http://www.Boost.org/more/discussion_policy.htm

[4] http://www.boost.org/libs/libraries.htm

[5] http://www.boost.org/libs/libraries.htm#Correctness