"...one of the most highly regarded and expertly designed C++ library projects in the world."
— Herb Sutter and Andrei Alexandrescu, C++ Coding Standards

Getting Started on Unix Variants

Index

1 Get Boost

The most reliable way to get a copy of Boost is to download a distribution from SourceForge:

1. Download `boost_1_52_0.tar.bz2`.

2. In the directory where you want to put the Boost installation, execute

```
tar --bzip2 -xf /path/to/boost_1_52_0.tar.bz2
```

Other Packages

RedHat, Debian, and other distribution packagers supply Boost library packages, however you may need to adapt these instructions if you use third-party packages, because their creators usually choose to break Boost up into several packages, reorganize the directory structure of the Boost distribution, and/or rename the library binaries.[1] If you have any trouble, we suggest using an official Boost distribution from SourceForge.

2 The Boost Distribution

This is a sketch of the resulting directory structure:

```
boost_1_52_0/ ..............The "boost root directory"
   index.htm .........A copy of www.boost.org starts here
   boost/ ........................All Boost Header files

   libs/ ............Tests, .cpps, docs, etc., by library
     index.html ........Library documentation starts here
     algorithm/
     any/
     array/
                    …more libraries…
   status/ .........................Boost-wide test suite
   tools/ ...........Utilities, e.g. Boost.Build, quickbook, bcp
   more/ ..........................Policy documents, etc.
   doc/ ..............A subset of all Boost library docs
```

It's important to note the following:

1. The path to the **boost root directory** (often `/usr/local/boost_1_52_0`) is sometimes referred to as `$BOOST_ROOT` in documentation and mailing lists .

2. To compile anything in Boost, you need a directory containing the `boost/` subdirectory in your `#include` path.

3. Since all of Boost's header files have the `.hpp` extension, and live in the

Header Organization

The organization of Boost library headers isn't entirely uniform, but most libraries follow a few patterns:

- Some older libraries and most very small libraries place all public headers directly into `boost/`.
- Most libraries' public headers live in a subdirectory of

`boost/` subdirectory of the boost root, your Boost `#include` directives will look like:

```
#include <boost/whatever.hpp>
```

or

```
#include "boost/whatever.hpp"
```

depending on your preference regarding the use of angle bracket includes.

4. Don't be distracted by the `doc/` subdirectory; it only contains a subset of the Boost documentation. Start with `libs/index.html` if you're looking for the whole enchilada.

`boost/`, named after the library. For example, you'll find the Python library's `def.hpp` header in

```
boost/python/def.hpp.
```

- Some libraries have an “aggregate header” in `boost/` that `#includes` all of the library's other headers. For example, Boost.Python's aggregate header is

```
boost/python.hpp.
```

- Most libraries place private headers in a subdirectory called `detail/`, or `aux_/`. Don't expect to find anything you can use in these directories.

3 Header-Only Libraries

The first thing many people want to know is, “how do I build Boost?” The good news is that often, there's nothing to build.

Nothing to Build?

Most Boost libraries are **header-only**: they consist *entirely of header files* containing templates and inline functions, and require no separately-compiled library binaries or special treatment when linking.

The only Boost libraries that *must* be built separately are:

- Boost.Filesystem
- Boost.GraphParallel
- Boost.IOStreams
- Boost.MPI
- Boost.ProgramOptions
- Boost.Python (see the Boost.Python build documentation before building and installing it)
- Boost.Regex
- Boost.Serialization
- Boost.Signals
- Boost.System
- Boost.Thread
- Boost.Wave

A few libraries have optional separately-compiled binaries:

- Boost.DateTime has a binary component that is only needed if you're using its `to_string`/`from_string` or serialization features, or if you're targeting Visual C++ 6.x or Borland.
- Boost.Graph also has a binary component that is only needed if you intend to parse GraphViz files.
- Boost.Math has binary components for the TR1 and C99 cmath functions.
- Boost.Random has a binary component which is only needed if you're using `random_device`.
- Boost.Test can be used in "header-only" or "separately compiled" mode, although **separate compilation is recommended for serious use**.

4 Build a Simple Program Using Boost

To keep things simple, let's start by using a header-only library. The following program reads a sequence of integers from standard input, uses Boost.Lambda to multiply each number by three, and writes them to standard output:

```
#include <boost/lambda/lambda.hpp>
#include <iostream>
#include <iterator>
#include <algorithm>

int main()
{
    using namespace boost::lambda;
    typedef std::istream_iterator<int> in;

    std::for_each(
        in(std::cin), in(), std::cout << (_1 * 3) << " " );
}
```

Copy the text of this program into a file called `example.cpp`.

Now, in the directory where you saved `example.cpp`, issue the following command:

```
c++ -I path/to/boost_1_52_0 example.cpp -o example
```

To test the result, type:

```
echo 1 2 3 | ./example
```

4.1 Errors and Warnings

Don't be alarmed if you see compiler warnings originating in Boost headers. We try to eliminate them, but doing so isn't always practical.[3] **Errors are another matter**. If you're seeing compilation errors at this point in the tutorial, check to be sure you've copied the example program correctly and that

you've correctly identified the Boost root directory.

5 Prepare to Use a Boost Library Binary

If you want to use any of the separately-compiled Boost libraries, you'll need to acquire library binaries.

5.1 Easy Build and Install

Issue the following commands in the shell (don't type `$`; that represents the shell's prompt):

```
$ cd path/to/boost_1_52_0
$ ./bootstrap.sh --help
```

Select your configuration options and invoke `./bootstrap.sh` again without the `--help` option. Unless you have write permission in your system's `/usr/local/` directory, you'll probably want to at least use

```
$ ./bootstrap.sh --prefix=path/to/installation/prefix
```

to install somewhere else. Also, consider using the `--show-libraries` and `--with-libraries=`*library-name-list* options to limit the long wait you'll experience if you build everything. Finally,

```
$ ./b2 install
```

will leave Boost binaries in the `lib/` subdirectory of your installation prefix. You will also find a copy of the Boost headers in the `include/` subdirectory of the installation prefix, so you can henceforth use that directory as an `#include` path in place of the Boost root directory.

skip to the next step

5.2 Or, Build Custom Binaries

If you're using a compiler other than your system's default, you'll need to use Boost.Build to create binaries.

You'll also use this method if you need a nonstandard build variant (see the Boost.Build documentation for more details).

> **Boost.CMake**
>
> There is also an experimental CMake build for boost, supported and distributed separately. See the Boost.CMake wiki page for more information.

5.2.1 Install Boost.Build

Boost.Build is a text-based system for developing, testing, and installing software. First, you'll need to build and install it. To do this:

1. Go to the directory `tools/build/v2/`.
2. Run `bootstrap.sh`
3. Run `b2 install --prefix=`*PREFIX* where *PREFIX* is the directory where you want Boost.Build to be installed
4. Add *PREFIX*`/bin` to your PATH environment variable.

5.2.2 Identify Your Toolset

First, find the toolset corresponding to your compiler in the following table (an up-to-date list is always available in the Boost.Build documentation).

> **Note**
>
> If you previously chose a toolset for the purposes of building b2, you should assume it won't work and instead choose newly from the table below.

Toolset Name	Vendor	Notes
`acc`	Hewlett Packard	Only very recent versions are known to work well with Boost
`borland`	Borland	
`como`	Comeau Computing	Using this toolset may require configuring another toolset to act as its backend
`darwin`	Apple Computer	Apple's version of the GCC toolchain with support for Darwin and MacOS X features such as frameworks.
`gcc`	The Gnu Project	Includes support for Cygwin and MinGW compilers.
`hp_cxx`	Hewlett Packard	Targeted at the Tru64 operating system.
`intel`	Intel	
`msvc`	Microsoft	
`sun`	Sun	Only very recent versions are known to work well with Boost.
`vacpp`	IBM	The VisualAge C++ compiler.

If you have multiple versions of a particular compiler installed, you can append the version number to the toolset name, preceded by a hyphen, e.g. `intel-9.0` or `borland-5.4.3`.

5.2.3 Select a Build Directory

Boost.Build will place all intermediate files it generates while building into the **build directory**. If your Boost root directory is writable, this step isn't strictly necessary: by default Boost.Build will create a `bin.v2/` subdirectory for that purpose in your current working directory.

5.2.4 Invoke `b2`

Change your current directory to the Boost root directory and invoke `b2` as follows:

```
b2 --build-dir=build-directory toolset=toolset-name   stage
```

For a complete description of these and other invocation options, please see the Boost.Build documentation.

For example, your session might look like this:

```
$ cd ~/boost_1_52_0
$ b2 --build-dir=/tmp/build-boost toolset=gcc stage
```

That will build static and shared non-debug multi-threaded variants of the libraries. To build all variants, pass the additional option, "`--build-type=complete`".

Building the special `stage` target places Boost library binaries in the `stage/lib/` subdirectory of the Boost tree. To use a different directory pass the `--stagedir=`*directory* option to `b2`.

> **Note**
>
> `b2` is case-sensitive; it is important that all the parts shown in **bold** type above be entirely lower-case.

For a description of other options you can pass when invoking `b2`, type:

```
b2 --help
```

In particular, to limit the amount of time spent building, you may be interested in:

- reviewing the list of library names with `--show-libraries`
- limiting which libraries get built with the `--with-`*library-name* or `--without-`*library-name* options
- choosing a specific build variant by adding `release` or `debug` to the command line.

Note

Boost.Build can produce a great deal of output, which can make it easy to miss problems. If you want to make sure everything is went well, you might redirect the output into a file by appending "`>build.log 2>&1`" to your command line.

5.3 Expected Build Output

During the process of building Boost libraries, you can expect to see some messages printed on the console. These may include

- Notices about Boost library configuration—for example, the Regex library outputs a message about ICU when built without Unicode support, and the Python library may be skipped without error (but with a notice) if you don't have Python installed.
- Messages from the build tool that report the number of targets that were built or skipped. Don't be surprised if those numbers don't make any sense to you; there are many targets per library.
- Build action messages describing what the tool is doing, which look something like:

```
toolset-name.c++ long/path/to/file/being/built
```

- Compiler warnings.

5.4 In Case of Build Errors

The only error messages you see when building Boost—if any—should be related to the IOStreams library's support of zip and bzip2 formats as described here. Install the relevant development packages for libz and libbz2 if you need those features. Other errors when building Boost libraries are cause for concern.

If it seems like the build system can't find your compiler and/or linker, consider setting up a `user-config.jam` file as described here. If that isn't your problem or the `user-config.jam` file doesn't work for you, please address questions about configuring Boost for your compiler to the Boost.Build mailing list.

6 Link Your Program to a Boost Library

To demonstrate linking with a Boost binary library, we'll use the following simple program that extracts the subject lines from emails. It uses the Boost.Regex library, which has a separately-compiled binary component.

```
#include <boost/regex.hpp>
#include <iostream>
#include <string>

int main()
{
```

```
        std::string line;
        boost::regex pat( "^Subject: (Re: |Aw: )*(.*)" );

        while (std::cin)
        {
            std::getline(std::cin, line);
            boost::smatch matches;
            if (boost::regex_match(line, matches, pat))
                std::cout << matches[2] << std::endl;
        }
    }
```

There are two main challenges associated with linking:

1. Tool configuration, e.g. choosing command-line options or IDE build settings.
2. Identifying the library binary, among all the build variants, whose compile configuration is compatible with the rest of your project.

There are two main ways to link to libraries:

A. You can specify the full path to each library:

```
$ c++ -I path/to/boost_1_52_0 example.cpp -o example \
   ~/boost/stage/lib/libboost_regex-gcc34-mt-d-1_36.a
```

B. You can separately specify a directory to search (with `-L`*directory*) and a library name to search for (with `-l`*library*,[2] dropping the filename's leading `lib` and trailing suffix (`.a` in this case):

```
$ c++ -I path/to/boost_1_52_0 example.cpp -o example \
   -L~/boost/stage/lib/ -lboost_regex-gcc34-mt-d-1_36
```

As you can see, this method is just as terse as method A for one library; it *really* pays off when you're using multiple libraries from the same directory. Note, however, that if you use this method with a library that has both static (`.a`) and dynamic (`.so`) builds, the system may choose one automatically for you unless you pass a special option such as `-static` on the command line.

In both cases above, the bold text is what you'd add to the command lines we explored earlier.

6.1 Library Naming

In order to choose the right binary for your build configuration you need to know how Boost binaries are named. Each library filename is composed of a common sequence of elements that describe how it was built. For example, `libboost_regex-vc71-mt-d-1_34.lib` can be broken down into the following elements:

`lib`
: *Prefix*: except on Microsoft Windows, every Boost library name begins with this string. On Windows, only ordinary static libraries use the `lib` prefix; import libraries and DLLs do not.[4]

boost_regex
Library name: all boost library filenames begin with `boost_`.

-vc71
Toolset tag: identifies the toolset and version used to build the binary.

-mt
Threading tag: indicates that the library was built with multithreading support enabled. Libraries built without multithreading support can be identified by the absence of `-mt`.

-d
ABI tag: encodes details that affect the library's interoperability with other compiled code. For each such feature, a single letter is added to the tag:

Key	Use this library when:	Boost.Build option
`s`	linking statically to the C++ standard library and compiler runtime support libraries.	runtime-link=static
`g`	using debug versions of the standard and runtime support libraries.	runtime-debugging=on
`y`	using a special debug build of Python.	python-debugging=on
`d`	building a debug version of your code.[5]	variant=debug
`p`	using the STLPort standard library rather than the default one supplied with your compiler.	stdlib=stlport

For example, if you build a debug version of your code for use with debug versions of the static runtime library and the STLPort standard library in “native iostreams” mode, the tag would be: `-sgdpn`. If none of the above apply, the ABI tag is ommitted.

-1_34
Version tag: the full Boost release number, with periods replaced by underscores. For example, version 1.31.1 would be tagged as "-1_31_1".

.lib
Extension: determined according to the operating system's usual convention. On most unix-style platforms the extensions are `.a` and `.so` for static libraries (archives) and shared libraries, respectively. On Windows, `.dll` indicates a shared library and `.lib` indicates a static or import library. Where supported by toolsets on unix variants, a full version extension is added (e.g. ".so.1.34") and a symbolic link to the library file, named without the trailing version number, will also be created.

6.2 Test Your Program

To test our subject extraction, we'll filter the following text file. Copy it out of your browser and save it as `jayne.txt`:

```
To: George Shmidlap
From: Rita Marlowe
Subject: Will Success Spoil Rock Hunter?
---
See subject.
```

If you linked to a shared library, you may need to prepare some platform-specific settings so that the system will be able to find and load it when your program is run. Most platforms have an environment variable to which you can add the directory containing the library. On many platforms (Linux, FreeBSD) that variable is `LD_LIBRARY_PATH`, but on MacOS it's `DYLD_LIBRARY_PATH`, and on Cygwin it's simply `PATH`. In most shells other than `csh` and `tcsh`, you can adjust the variable as follows (again, don't type the `$`—that represents the shell prompt):

```
$ VARIABLE_NAME=path/to/lib/directory:${VARIABLE_NAME}
$ export VARIABLE_NAME
```

On `csh` and `tcsh`, it's

```
$ setenv VARIABLE_NAME path/to/lib/directory:${VARIABLE_NAME}
```

Once the necessary variable (if any) is set, you can run your program as follows:

```
$ path/to/compiled/example < path/to/jayne.txt
```

The program should respond with the email subject, “Will Success Spoil Rock Hunter?”

7 Conclusion and Further Resources

This concludes your introduction to Boost and to integrating it with your programs. As you start using Boost in earnest, there are surely a few additional points you'll wish we had covered. One day we may have a “Book 2 in the Getting Started series” that addresses them. Until then, we suggest you pursue the following resources. If you can't find what you need, or there's anything we can do to make this document clearer, please post it to the Boost Users' mailing list.

- Boost.Build reference manual
- Boost Users' mailing list
- Boost.Build mailing list
- Index of all Boost library documentation

Onward

Good luck, and have fun!

—the Boost Developers

[1] If developers of Boost packages would like to work with us to make sure these instructions can be used with their packages, we'd be glad to help. Please make your interest known to the Boost developers' list.

[2] That option is a dash followed by a lowercase “L” character, which looks very much like a numeral 1 in some fonts.

[3] Remember that warnings are specific to each compiler implementation. The developer of a given Boost library might not have access to your compiler. Also, some warnings are extremely difficult to eliminate in generic code, to the point where it's not worth the trouble. Finally, some compilers don't have any source code mechanism for suppressing warnings.

[4] This convention distinguishes the static version of a Boost library from the import library for an identically-configured Boost DLL, which would otherwise have the same name.

[5] These libraries were compiled without optimization or inlining, with full debug symbols enabled, and without `NDEBUG` `#defined`. Although it's true that sometimes these choices don't affect binary compatibility with other compiled code, you can't count on that with Boost libraries.

[6] This feature of STLPort is deprecated because it's impossible to make it work transparently to the user; we don't recommend it.