BCP

John Maddock

Table of Contents

Overview

The bcp utility is a tool for extracting subsets of Boost, it's useful for Boost authors who want to distribute their library separately from Boost, and for Boost users who want to distribute a subset of Boost with their application.

bcp can also report on which parts of Boost your code is dependent on, and what licences are used by those dependencies.

Examples

```
bcp scoped_ptr /foo
```

Copies boost/scoped_ptr.hpp and dependencies to /foo.

```
bcp boost/regex.hpp /foo
```

Copies boost/regex.hpp and all dependencies including the regex source code (in libs/regex/src) and build files (in libs/regex/build) to /foo. Does not copy the regex documentation, test, or example code. Also does not copy the Boost.Build system.

```
bcp regex /foo
```

Copies the full regex lib (in libs/regex) including dependencies (such as the boost.test source required by the regex test programs) to /foo. Does not copy the Boost.Build system.

```
bcp --namespace=myboost --namespace-alias regex config build /foo
```

Copies the full regex lib (in libs/regex) plus the config lib (libs/config) and the build system (tools/build) to /foo including all the dependencies. Also renames the boost namespace to *myboost* and changes the filenames of binary libraries to begin with the prefix "myboost" rather than "boost". The --namespace-alias option makes `namespace boost` an alias of the new name.

```
bcp --scan --boost=/boost foo.cpp bar.cpp boost
```

Scans the [non-boost] files foo.cpp and bar.cpp for boost dependencies and copies those dependencies to the sub-directory boost.

```
bcp --report regex.hpp boost-regex-report.html
```

Creates a HTML report called boost-regex-report.html for the boost module regex.hpp. The report contains license information, author details, and file dependencies.

Syntax

Behaviour Selection

```
bcp --list [options] module-list
```

Outputs a list of all the files in module-list including dependencies.

```
bcp [options] module-list output-path
```

Copies all the files found in module-list to output-path

```
bcp --report [options] module-list html-file
```

Outputs a html report file containing:

- All the licenses in effect, plus the files using each license, and the copyright holders using each license.
- Any files with no recognizable license (please report these to the boost mailing lists).
- Any files with no recognizable copyright holders (please report these to the boost mailing lists).
- All the copyright holders and the files on which they hold copyright.
- File dependency information - indicates the reason for the inclusion of any particular file in the dependencies found.

Options

```
--boost=path
```

Sets the location of the boost tree to path. If this option is not provided then the current path is assumed to be the root directory of the Boost tree.

```
--namespace=newname ↵
```

When copying files, all occurances of the boost namespace will get renamed to "newname". Also renames Boost binaries to use "newname" rather than "boost" as a prefix.

Often used in conjunction with the --namespace-alias option, this allows two different Boost versions to be used in the same program, but not in the same translation unit.

```
--namespace-alias
```

When used in conjunction with the --namespace option, then `namespace boost` will be declared as an alias of the new namespace name. This allows existing code that relies on Boost code being in `namespace boost` to compile unchanged, while retaining the "strong versioning" that can be achieved with a namespace change.

```
--scan
```

Treats the module list as a list of (probably non-boost) files to scan for boost dependencies, the files listed in the module list are not copied (or listed), only the boost files upon which they depend.

```
--svn
```

Only copy files under svn version control.

```
--unix-lines
```

Make sure that all copied files use Unix style line endings.

module-list

When the --scan option is not used then a list of boost files or library names to copy, this can be:

1. The name of a tool: for example "build" will find "tools/build".
2. The name of a library: for example "regex".
3. The title of a header: for example "scoped_ptr" will find "boost/scoped_ptr.hpp".
4. The name of a header: for example "scoped_ptr.hpp" will find "boost/scoped_ptr.hpp".
5. The name of a file: for example "boost/regex.hpp".

When the --scan option is used, then a list of (probably non-boost) files to scan for boost dependencies, the files in the module list are not therefore copied/listed.

output-path

The path to which files will be copied (this path must exist).

Dependencies

File dependencies are found as follows:

- C++ source files are scanned for #includes, all #includes present in the boost source tree will then be scanned for their dependencies and so on.
- C++ source files are associated with the name of a library, if that library has source code (and possibly build data), then include that source in the dependencies.
- C++ source files are checked for dependencies on Boost.test (for example to see if they use cpp_main as an entry point).
- HTML files are scanned for immediate dependencies (images and style sheets, but not links).

It should be noted that in practice bcp can produce a rather "fat" list of dependencies, reasons for this include:

- It searches for library names first, so using "regex" as a name will give you everything in the libs/regex directory and everything that depends on. This can be a long list as all the regex test and example programs will get scanned for their dependencies. If you want a more minimal list, then try using the names of the headers you are actually including, or use the --scan option to scan your source code.
- If you include the header of a library with separate source, then you get that libraries source and all it's dependencies. This is deliberate and in general those extra dependencies are needed.
- When you include a header, bcp doesn't know what compiler you're using, so it follows all possible preprocessor paths. If you're distributing a subset of Boost with you're application then that is what you want to have happen in general.

The last point above can result in a substantial increase in the number of headers found compared to most peoples expectations. For example bcp finds 274 header dependencies for boost/shared_ptr.hpp: by running bcp in report mode we can see why all these headers have been found as dependencies:

- All of the Config library headers get included (52 headers, would be about 6 for one compiler only).
- A lot of MPL and type traits code that includes workarounds for broken compilers that you may or may not need. Tracing back through the code shows that most of these aren't needed unless the user has defined BOOST_SP_USE_QUICK_ALLOCATOR, however bcp isn't aware of whether that preprocessor path will be taken or not, so the headers get included just in case. This adds about 48 headers (type traits), plus another 49 from MPL.
- The Preprocessor library gets used heavily by MPL: this adds another 96 headers.
- The Shared Pointer library contains a lot of platform specific code, split up into around 22 headers: normally your compiler would need only a couple of these files.

As you can see the number of dependencies found are much larger than those used by any single compiler, however if you want to distribute a subset of Boost that's usable in any configuration, by any compiler, on any platform then that's exactly what you need. If you want to figure out which Boost headers are being used by your specific compiler then the best way to find out is to prepocess the code and scan the output for boost header includes. You should be aware that the result will be very platform and compiler specific, and may not contain all the headers needed if you so much as change a compiler switch (for example turn on threading support).