Boost.Config

Vesa Karvonen, John Maddock Beman Dawes

Table of Contents

Configuring Boost for Your Platform

Using the default boost configuration

Boost comes already configured for most common compilers and platforms; you should be able to use boost "as is". Since the compiler is configured separately from the standard library, the default configuration should work even if you replace the compiler's standard library with a third-party standard library (like STLport).

Using boost "as is" without trying to reconfigure is the recommended method for using boost. You can, however, run the configure script if you want to, and there are regression tests provided that allow you to test the current boost configuration with your particular compiler setup.

Boost library users can request support for additional compilers or platforms by visiting our Tracker and submitting a support request.

The <boost/config.hpp> header

Boost library implementations access configuration macros via

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

While Boost library users are not required to include that file directly, or use those configuration macros, such use is acceptable. The configuration macros are documented as to their purpose, usage, and limitations which makes them usable by both Boost library and user code.

Boost informational or helper macros are designed for use by Boost users as well as for our own internal use. Note however, that the feature test and defect test macros were designed for internal use by Boost libraries, not user code, so they can change at any time (though no gratuitous changes are made to them). Boost library problems resulting from changes to the configuration macros are caught by the Boost regression tests, so the Boost libraries are updated to account for those changes. By contrast, Boost library user code can be adversely affected by changes to the macros without warning. The best way to keep abreast of changes to the macros used in user code is to monitor the discussions on the Boost developers list.

Using the configure script

Important

This configure script only sets up the Boost headers for use with a particular compiler. It has no effect on Boost.Build, or how the libraries are built.

If you know that boost is incorrectly configured for your particular setup, and you are on a UNIX like platform, then you may want to try and improve things by running the boost configure script. From a shell command prompt you will need to cd into <*boost-root*>`/libs/config/` and type:

```
sh ./configure
```

you will see a list of the items being checked as the script works its way through the regression tests. Note that the configure script only really auto-detects your compiler if it's called g++, c++ or CC. If you are using some other compiler you will need to set one or more of the following environment variables:

Variable	Description
CXX	The name of the compiler, for example `c++`.
CXXFLAGS	The compiler flags to use, for example `-O2`.
LDFLAGS	The linker flags to use, for example `-L/mypath`.
LIBS	Any libraries to link in, for example `-lpthread`.

For example to run the configure script with HP aCC, you might use something like:

```
export CXX="aCC"
export CXXFLAGS="-Aa -DAportable -D__HPACC_THREAD_SAFE_RB_TREE \
   -DRWSTD_MULTI_THREAD -DRW_MULTI_THREAD -D_REENTRANT -D_THREAD_SAFE"
export LDFLAGS="-DAportable"
export LIBS="-lpthread"
sh ./configure
```

However you run the configure script, when it finishes you will find a new header `-user.hpp-` located in the <*boost-root*>`/libs/config/` directory. **Note that configure does not install this header into your boost include path by default**. This header contains all the options generated by the configure script, plus a header-section that contains the user settable options from the default version of <boost/config/user.hpp> (located under <*boost-root*>`/boost/config/`). There are two ways you can use this header:

- **Option 1:** copy the header into <*boost-root*>`/boost/config/` so that it replaces the default user.hpp provided by boost. This option allows only one configure-generated setup; boost developers should avoid this option, as it incurs the danger of accidentally committing a configure-modified <boost/config/user.hpp> to the cvs repository (something you will not be thanked for!).

- **Option 2:** give the header a more memorable name, and place it somewhere convenient; then, define the macro `BOOST_USER_CONFIG` to point to it. For example create a new sub-directory <*boost-root*>`/boost/config/user/`, and copy the header there; for example as `multithread-gcc-config.hpp`. Then, when compiling add the command line option: `-DBOOST_USER_CONFIG="<boost/config/user/multithread-gcc-config.hpp>"`, and boost will use the new configuration header. This option allows you to generate more than one configuration header, and to keep them separate from the boost source - so that updates to the source do not interfere with your configuration.

User settable options

There are some configuration-options that represent user choices, rather than compiler defects or platform specific options. These are listed in `<boost/config/user.hpp>` and at the start of a configure-generated `user.hpp` header. You can define these on the command line, or by editing `<boost/config/user.hpp>`, they are listed in the following table:

Macro	Description
BOOST_USER_CONFIG	When defined, it should point to the name of the user configuration file to include prior to any boost configuration files. When not defined, defaults to <boost/config/user.hpp>.
BOOST_COMPILER_CONFIG	When defined, it should point to the name of the compiler configuration file to use. Defining this cuts out the compiler selection logic, and eliminates the dependency on the header containing that logic. For example if you are using gcc, then you could define BOOST_COMPILER_CONFIG to <boost/config/compiler/gcc.hpp>.
BOOST_STDLIB_CONFIG	When defined, it should point to the name of the standard library configuration file to use. Defining this cuts out the standard library selection logic, and eliminates the dependency on the header containing that logic. For example if you are using STLport, then you could define BOOST_STDLIB_CONFIG to <boost/config/stdlib/stlport.hpp>.
BOOST_PLATFORM_CONFIG	When defined, it should point to the name of the platform configuration file to use. Defining this cuts out the platform selection logic, and eliminates the dependency on the header containing that logic. For example if you are compiling on linux, then you could define BOOST_PLATFORM_CONFIG to <boost/config/platform/linux.hpp>.
BOOST_NO_COMPILER_CONFIG	When defined, no compiler configuration file is selected or included, define when the compiler is fully conformant with the standard, or where the user header (see BOOST_USER_CONFIG), has had any options necessary added to it, for example by an autoconf generated configure script.
BOOST_NO_STDLIB_CONFIG	When defined, no standard library configuration file is selected or included, define when the standard library is fully conformant with the standard, or where the user header (see BOOST_USER_CONFIG), has had any options necessary added to it, for example by an autoconf generated configure script.
BOOST_NO_PLATFORM_CONFIG	When defined, no platform configuration file is selected or included, define when the platform is fully conformant with the standard (and has no useful extra features), or where the user header (see BOOST_USER_CONFIG), has had any options necessary added to it, for example by an autoconf generated configure script.
BOOST_NO_CONFIG	Equivalent to defining all of BOOST_NO_COMPILER_CONFIG, BOOST_NO_STDLIB_CONFIG and BOOST_NO_PLATFORM_CONFIG.
BOOST_STRICT_CONFIG	The normal behavior for compiler versions that are newer than the last known version, is to assume that they have all the same defects as the last known version. By setting this define, then compiler versions that are newer than the last known version are assumed to be fully conforming with the standard. This is probably most useful for boost developers or testers, and for those who want to use boost to test beta compiler versions.
BOOST_ASSERT_CONFIG	When this flag is set, if the config finds anything unknown, then it will stop with a #error rather than continue. Boost regression testers should set this define, as should anyone who wants to quickly check whether boost is supported on their platform.
BOOST_DISABLE_THREADS	When defined, disables threading support, even if the compiler in its current translation mode supports multiple threads.
BOOST_DISABLE_WIN32	When defined, disables the use of Win32 specific API's, even when these are available. Also has the effect of setting BOOST_DISABLE_THREADS unless BOOST_HAS_PTHREADS is set. This option may be set automatically by the config system when it detects that the compiler is in "strict mode".

Macro	Description
`BOOST_DISABLE_ABI_HEADERS`	Stops boost headers from including any prefix/suffix headers that normally control things like struct packing and alignment.
`BOOST_ABI_PREFIX`	A prefix header to include in place of whatever boost.config would normally select, any replacement should set up struct packing and alignment options as required.
`BOOST_ABI_SUFFIX`	A suffix header to include in place of whatever boost.config would normally select, any replacement should undo the effects of the prefix header.
`BOOST_ALL_DYN_LINK`	Forces all libraries that have separate source, to be linked as dll's rather than static libraries on Microsoft Windows (this macro is used to turn on `__declspec(dllimport)` modifiers, so that the compiler knows which symbols to look for in a dll rather than in a static library). Note that there may be some libraries that can only be statically linked (Boost.Test for example) and others which may only be dynamically linked (Boost.Threads for example), in these cases this macro has no effect.
`BOOST_`*WHATEVER*`_DYN_LINK`	Forces library "whatever" to be linked as a dll rather than a static library on Microsoft Windows: replace the *WHATEVER* part of the macro name with the name of the library that you want to dynamically link to, for example use `BOOST_DATE_TIME_DYN_LINK` or `BOOST_REGEX_DYN_LINK` etc (this macro is used to turn on `__declspec(dllimport)` modifiers, so that the compiler knows which symbols to look for in a dll rather than in a static library). Note that there may be some libraries that can only be statically linked (Boost.Test for example) and others which may only be dynamically linked (Boost.Threads for example), in these cases this macro is unsupported.
`BOOST_ALL_NO_LIB`	Tells the config system not to automatically select which libraries to link against. Normally if a compiler supports #pragma lib, then the correct library build variant will be automatically selected and linked against, simply by the act of including one of that library's headers. This macro turns that feature off.
`BOOST_`*WHATEVER*`_NO_LIB`	Tells the config system not to automatically select which library to link against for library "whatever", replace *WHATEVER* in the macro name with the name of the library; for example `BOOST_DATE_TIME_NO_LIB` or `BOOST_REGEX_NO_LIB`. Normally if a compiler supports `#pragma lib`, then the correct library build variant will be automatically selected and linked against, simply by the act of including one of that library's headers. This macro turns that feature off.
`BOOST_LIB_DIAGNOSTIC`	Causes the auto-linking code to output diagnostic messages indicating the name of the library that is selected for linking.
`BOOST_LIB_TOOLSET`	Overrides the name of the toolset part of the name of library being linked to; note if defined this must be defined to a quoted string literal, for example "abc".

Advanced configuration usage

By setting various macros on the compiler command line or by editing <boost/config/user.hpp>, the boost configuration setup can be optimised in a variety of ways.

Boost's configuration is structured so that the user-configuration is included first (defaulting to <boost/config/user.hpp> if `BOOST_USER_CONFIG` is not defined). This sets up any user-defined policies, and gives the user-configuration a chance to influence what happens next.

Next the compiler, standard library, and platform configuration files are included. These are included via macros (`BOOST_COMPILER_CONFIG` etc, see user settable macros), and if the corresponding macro is undefined then a separate header that detects which compiler/standard library/platform is in use is included in order to set these. The config can be told to ignore these headers altogether

if the corresponding BOOST_NO_XXX macro is set (for example BOOST_NO_COMPILER_CONFIG to disable including any compiler configuration file - see user settable macros).

Finally the boost configuration header, includes <boost/config/suffix.hpp>; this header contains any boiler plate configuration code - for example where one boost macro being set implies that another must be set also.

The following usage examples represent just a few of the possibilities:

Example 1, creating our own frozen configuration

Lets suppose that we're building boost with Visual C++ 6, and STLport 4.0. Lets suppose also that we don't intend to update our compiler or standard library any time soon. In order to avoid breaking dependencies when we update boost, we may want to "freeze" our configuration headers, so that we only have to rebuild our project if the boost code itself has changed, and not because the boost config has been updated for more recent versions of Visual C++ or STLport. We'll start by realising that the configuration files in use are: <boost/config/compiler/visualc.hpp> for the compiler, <boost/config/stdlib/stlport.hpp> for the standard library, and <boost/config/platform/win32.hpp> for the platform. Next we'll create our own private configuration directory: boost/config/mysetup/, and copy the configuration files into there. Finally, open up <boost/config/user.hpp> and edit the following defines:

```
#define BOOST_COMPILER_CONFIG "boost/config/mysetup/visualc.hpp"
#define BOOST_STDLIB_CONFIG "boost/config/mysetup/stlport.hpp"
#define BOOST_USER_CONFIG "boost/config/mysetup/win32.hpp"
```

Now when you use boost, its configuration header will go straight to our "frozen" versions, and ignore the default versions, you will now be insulated from any configuration changes when you update boost. This technique is also useful if you want to modify some of the boost configuration files; for example if you are working with a beta compiler release not yet supported by boost.

Example 2: skipping files that you don't need

Lets suppose that you're using boost with a compiler that is fully conformant with the standard; you're not interested in the fact that older versions of your compiler may have had bugs, because you know that your current version does not need any configuration macros setting. In a case like this, you can define BOOST_NO_COMPILER_CONFIG either on the command line, or in <boost/config/user.hpp>, and miss out the compiler configuration header altogether (actually you miss out two headers, one which works out what the compiler is, and one that configures boost for it). This has two consequences: the first is that less code has to be c ompiled, and the second that you have removed a dependency on two boost headers.

Example 3: using configure script to freeze the boost configuration

If you are working on a unix-like platform then you can use the configure script to generate a "frozen" configuration based on your current compiler setup - see using the configure script for more details.

Testing the boost configuration

The boost configuration library provides a full set of regression test programs under the *<boost-root>*/boost/config/ test/ sub-directory:

File	Description
`config_info.cpp`	Prints out a detailed description of your compiler/standard library/platform setup, plus your current boost configuration. The information provided by this program useful in setting up the boost configuration files. If you report that boost is incorrectly configured for your compiler/library/platform then please include the output from this program when reporting the changes required.
`config_test.cpp`	A monolithic test program that includes most of the individual test cases. This provides a quick check to see if boost is correctly configured for your compiler/library/platform.
`limits_test.cpp`	Tests your standard library's `std::numeric_limits` implementation (or its boost provided replacement if `BOOST_NO_LIMITS` is defined). This test file fails with most versions of numeric_limits, mainly due to the way that some compilers treat NAN's and infinity.
`no_*pass.cpp`	Individual compiler defect test files. Each of these should compile, if one does not then the corresponding `BOOST_NO_XXX` macro needs to be defined - see each test file for specific details.
`no_*fail.cpp`	Individual compiler defect test files. Each of these should not compile, if one does then the corresponding `BOOST_NO_XXX` macro is defined when it need not be - see each test file for specific details.
`has_*pass.cpp`	Individual feature test files. If one of these does not compile then the corresponding `BOOST_HAS_XXX` macro is defined when it should not be - see each test file for specific details.
`has_*fail.cpp`	Individual feature test files. If one of these does compile then the corresponding `BOOST_HAS_XXX` macro can be safely defined - see each test file for specific details.

Although you can run the configuration regression tests as individual test files, there are rather a lot of them, so there are a couple of shortcuts to help you out:

If you have built the boost regression test driver, then you can use this to produce a nice html formatted report of the results using the supplied test file.

Alternatively you can run the configure script like this:

```
./configure --enable-test
```

in which case the script will test the current configuration rather than creating a new one from scratch.

If you are reporting the results of these tests for a new platform/library/compiler then please include a log of the full compiler output, the output from `config_info.cpp`, and the pass/fail test results.

Boost Macro Reference

Macros that describe defects

The following macros all describe features that are required by the C++ standard, if one of the following macros is defined, then it represents a defect in the compiler's conformance with the standard.

Macro	Section	Description
BOOST_BCB_PARTIAL_SPECIALIZATION_BUG	Compiler	The compiler exibits certain partial specialisation bug - probably Borland C++ Builder specific.
BOOST_FUNCTION_SCOPE_USING_DECLARATION_BREAKS_ADL	Compiler	Argument dependent lookup fails if there is a using declaration for the symbol being looked up in the current scope. For example, using `boost::get_pointer`; prevents ADL from finding overloads of `get_pointer` in namespaces nested inside boost (but not elsewhere). Probably Borland specific.
BOOST_NO_ADL_BARRIER	Compiler	The compiler locates and searches namespaces that it should *not* in fact search when performing argument dependent lookup.
BOOST_NO_ARGUMENT_DEPENDENT_LOOKUP	Compiler	Compiler does not implement argument-dependent lookup (also named Koenig lookup); see std::3.4.2 [basic.koenig.lookup]
BOOST_NO_AUTO_PTR	Standard library	If the compiler / library supplies non-standard or broken `std::auto_ptr`.
BOOST_NO_CTYPE_FUNCTIONS	Platform	The Platform does not provide functions for the character-classifying operations `<ctype.h>` and `<cctype>`, only macros.
BOOST_NO_CV_SPECIALIZATIONS	Compiler	If template specialisations for cv-qualified types conflict with a specialisation for a cv-unqualififed type.
BOOST_NO_CV_VOID_SPECIALIZATIONS	Compiler	If template specialisations for cv-void types conflict with a specialisation for void.
BOOST_NO_CWCHAR	Platform	The Platform does not provide `<wchar.h>` and `<cwchar>`.
BOOST_NO_CWCTYPE	Platform	The Platform does not provide `<wctype.h>` and `<cwctype>`.

BOOST_NO_DEPENDENT_NESTED_DERIVATIONS	Compiler	The compiler fails to compile a nested class that has a dependent base class:
		`template<typename T>`
		`struct foo : {`
		`  template<typename U>`
		`  struct bar : public U {};`
		`};`

BOOST_NO_DEPENDENT_TYPES_IN_TEMPLATE_VALUE_PARAMETERS	Compiler	Template value parameters cannot have a dependent type, for example:
		`template<class T, typename T::type value>`
		`class X { ... };`

BOOST_NO_EXCEPTION_STD_NAMESPACE	Standard Library	The standard library does not put some or all of the contents of `<exception>` in namespace std.

Macro	Section	Description
BOOST_NO_EXCEPTIONS	Compiler	The compiler does not support exception handling (this setting is typically required by many C++ compilers for embedded platforms). Note that there is no requirement for boost libraries to honor this configuration setting - indeed doing so may be impossible in some cases. Those libraries that do honor this will typically abort if a critical error occurs - you have been warned!
BOOST_NO_EXPLICIT_FUNCTION_TEMPLATE_ARGUMENTS	Compiler	Can only use deduced template arguments when calling function template instantiations.

BOOST_NO_FUNCTION_TEMPLATE_ORDERING	Compiler	The compiler does not perform function template ordering or its function template ordering is incorrect.
		`// #1`
		`template<class T> void f(T);`
		
		`// #2`
		`template<class T,class U> void f(T(*)(U));`
		
		`void bar(int);`
		
		`f(&bar); // should choose #2.`

BOOST_NO_INCLASS_MEMBER_INITIALIZATION	Compiler	Compiler violates std::9.4.2/4.
BOOST_NO_INTRINSIC_WCHAR_T	Compiler	The C++ implementation does not provide `wchar_t`, or it is really a synonym for another integral type. Use this symbol to decide whether it is appropriate to explicitly specialize a template on `wchar_t` if there is already a specialization for other integer types.
BOOST_NO_IOSFWD	std lib	The standard library lacks `<iosfwd>`.
BOOST_NO_IOSTREAM	std lib	The standard library lacks `<iostream>`, `<istream>` or `<ostream>`.
BOOST_NO_IS_ABSTRACT	Compiler	The C++ compiler does not support SFINAE with abstract types, this is covered by Core Language DR337, but is not part of the current standard. Fortunately most compilers that support SFINAE also support this DR.
BOOST_NO_LIMITS	Standard library	The C++ implementation does not provide the `<limits>` header. Never check for this symbol in library code; always include `<boost/limits.hpp>`, which guarantees to provide `std::numeric_limits`.
BOOST_NO_LIMITS_COMPILE_TIME_CONSTANTS	Standard library	Constants such as `numeric_limits<T>::is_signed` are not available for use at compile-time.
BOOST_NO_LONG_LONG_NUMERIC_LIMITS	Standard library	There is no specialization for `numeric_limits<long long>` and `numeric_limits<unsigned long long>`. `<boost/limits.hpp>` will then add these specializations as a standard library "fix" only if the compiler supports the `long long` datatype.

Macro	Section	Description
BOOST_NO_MEMBER_FUNCTION_SPECIALIZATIONS	Compiler	The compiler does not support the specialization of individual member functions of template classes.
BOOST_NO_MEMBER_TEMPLATE_KEYWORD	Compiler	If the compiler supports member templates, but not the template keyword when accessing member template classes.
BOOST_NO_MEMBER_TEMPLATE_FRIENDS	Compiler	Member template friend syntax (`template<class P> friend class frd;`) described in the C++ Standard, 14.5.3, not supported.
BOOST_NO_MEMBER_TEMPLATES	Compiler	Member template functions not fully supported.
BOOST_NO_MS_INT64_NUMERIC_LIMITS	Standard library	There is no specialization for `numeric_limits<__int64>` and `numeric_limits<unsigned __int64>`. `<boost/limits.hpp>` will then add these specializations as a standard library "fix", only if the compiler supports the `__int64` datatype.
BOOST_NO_NESTED_FRIENDSHIP	Compiler	Compiler doesn't allow a nested class to access private members of its containing class. Probably Borland/CodeGear specific.
BOOST_NO_OPERATORS_IN_NAMESPACE	Compiler	Compiler requires inherited operator friend functions to be defined at namespace scope, then using'ed to boost. Probably GCC specific. See `<boost/operators.hpp>` for example.
BOOST_NO_PARTIAL_SPECIALIZATION_IMPLICIT_DEFAULT_ARGS	Compiler	The compiler does not correctly handle partial specializations which depend upon default arguments in the primary template.
BOOST_NO_POINTER_TO_MEMBER_CONST	Compiler	The compiler does not correctly handle pointers to const member functions, preventing use of these in overloaded function templates. See `<boost/functional.hpp>` for example.
BOOST_NO_POINTER_TO_MEMBER_TEMPLATE_PARAMETERS	Compiler	Pointers to members don't work when used as template parameters.
BOOST_NO_PRIVATE_IN_AGGREGATE	Compiler	The compiler misreads 8.5.1, treating classes as non-aggregate if they contain private or protected member functions.
BOOST_NO_RTTI	Compiler	The compiler may (or may not) have the typeid operator, but RTTI on the dynamic type of an object is not supported.
BOOST_NO_SFINAE	Compiler	The compiler does not support the "Substitution Failure Is Not An Error" meta-programming idiom.
BOOST_NO_STD_ALLOCATOR	Standard library	The C++ standard library does not provide a standards conforming `std::allocator`.
BOOST_NO_STD_DISTANCE	Standard library	The platform does not have a conforming version of `std::distance`.
BOOST_NO_STD_ITERATOR	Standard library	The C++ implementation fails to provide the `std::iterator` class.
BOOST_NO_STD_ITERATOR_TRAITS	Standard library	The compiler does not provide a standard compliant implementation of `std::iterator_traits`. Note that the compiler may still have a non-standard implementation.
BOOST_NO_STD_LOCALE	Standard library	The standard library lacks `std::locale`.

Macro	Section	Description
BOOST_NO_STD_MESSAGES	Standard library	The standard library lacks a conforming std::messages facet.
BOOST_NO_STD_MIN_MAX	Standard library	The C++ standard library does not provide the min() and max() template functions that should be in <algorithm>.
BOOST_NO_STD_OUTPUT_ITERATOR_ASSIGN	Standard library	Defined if the standard library's output iterators are not assignable.
BOOST_NO_STD_TYPEINFO	Standard library	The <typeinfo> header declares type_info in the global namespace instead of namespace std.
BOOST_NO_STD_USE_FACET	Standard library	The standard library lacks a conforming std::use_facet.
BOOST_NO_STD_WSTREAMBUF	Standard library	The standard library's implementation of std::basic_streambuf<wchar_t> is either missing, incomplete, or buggy.
BOOST_NO_STD_WSTRING	Standard library	The standard library lacks std::wstring.
BOOST_NO_STDC_NAMESPACE	Compiler, Platform	The contents of C++ standard headers for C library functions (the <c...> headers) have not been placed in namespace std. This test is difficult - some libraries "fake" the std C functions by adding using declarations to import them into namespace std, unfortunately they don't necessarily catch all of them...
BOOST_NO_STRINGSTREAM	Standard library	The C++ implementation does not provide the <sstream> header.
BOOST_NO_SWPRINTF	Platform	The platform does not have a conforming version of swprintf.
BOOST_NO_TEMPLATE_PARTIAL_SPECIALIZATION	Compiler	Class template partial specialization (14.5.4 [temp.class.spec]) not supported.
BOOST_NO_TEMPLATED_IOSTREAMS	Standard library	The standard library does not provide templated iostream classes.
BOOST_NO_TEMPLATED_ITERATOR_CONSTRUCTORS	Standard library	The standard library does not provide templated iterator constructors for its containers.
BOOST_NO_TEMPLATE_TEMPLATES	Compiler	The compiler does not support template template parameters.
BOOST_NO_TYPEID	Compiler	The compiler does not support the typeid operator at all.
BOOST_NO_TYPENAME_WITH_CTOR	Compiler	The typename keyword cannot be used when creating a temporary of a Dependent type.
BOOST_NO_UNREACHABLE_RETURN_DETECTION	Compiler	If a return is unreachable, then no return statement should be required, however some compilers insist on it, while other issue a bunch of warnings if it is in fact present.
BOOST_NO_USING_DECLARATION_OVERLOADS_FROM_TYPENAME_BASE	Compiler	The compiler will not accept a using declaration that brings a function from a typename used as a base class into a derived class if functions of the same name are present in the derived class.

Macro	Section	Description
BOOST_NO_USING_TEMPLATE	Compiler	The compiler will not accept a using declaration that imports a template class or function from another namespace. Originally a Borland specific problem with imports to/from the global namespace, extended to MSVC6 which has a specific issue with importing template classes (but not functions).

BOOST_NO_VOID_RETURNS	Compiler	The compiler does not allow a void function to return the result of calling another void function.
		`void f() {}`
		`void g() { return f(); }`

Macros that describe optional features

The following macros describe features that are not required by the C++ standard. The macro is only defined if the feature is present.

Macro	Section	Description
BOOST_HAS_BETHREADS	Platform	The platform supports BeOS style threads.
BOOST_HAS_CLOCK_GETTIME	Platform	The platform has the POSIX API clock_gettime.
BOOST_HAS_DECLSPEC	Compiler	The compiler uses __declspec(dllexport) and __declspec(dllimport) to export/import symbols from dll's.
BOOST_HAS_DIRENT_H	Platform	The platform has the POSIX header <dirent.h>.
BOOST_HAS_EXPM1	Platform	The platform has the functions expm1, expm1f and expm1l in <math.h>
BOOST_HAS_FTIME	Platform	The platform has the Win32 API GetSystemTimeAsFileTime.
BOOST_HAS_GETTIMEOFDAY	Platform	The platform has the POSIX API gettimeofday.
BOOST_HAS_HASH	Standard library	The C++ implementation provides the (SGI) hash_set and hash_map classes. When defined, BOOST_HASH_SET_HEADER and BOOST_HASH_LIST_HEADER will contain the names of the header needed to access hash_set and hash_map; BOOST_STD_EXTENSION_NAMESPACE will provide the namespace in which the two class templates reside.
BOOST_HAS_LOG1P	Platform	The platform has the functions log1p, log1pf and log1pl in <math.h>.
BOOST_HAS_MACRO_USE_FACET	Standard library	The standard library lacks a conforming std::use_facet, but has a macro _USE(loc, Type) that does the job. This is primarily for the Dinkumware std lib.
BOOST_HAS_MS_INT64	Compiler	The compiler supports the __int64 data type.
BOOST_HAS_NANOSLEEP	Platform	The platform has the POSIX API nanosleep.
BOOST_HAS_NL_TYPES_H	Platform	The platform has an <nl_types.h>.
BOOST_HAS_NRVO	Compiler	Indicated that the compiler supports the named return value optimization (NRVO). Used to select the most efficient implementation for some function. See <boost/operators.hpp> for example.
BOOST_HAS_PARTIAL_STD_ALLOCATOR	Standard Library	The standard library has a partially conforming std::allocator class, but without any of the member templates.
BOOST_HAS_PTHREAD_DELAY_NP	Platform	The platform has the POSIX API pthread_delay_np.
BOOST_HAS_PTHREAD_MUTEXATTR_SETTYPE	Platform	The platform has the POSIX API pthread_mutexattr_settype.
BOOST_HAS_PTHREAD_YIELD	Platform	The platform has the POSIX API pthread_yield.
BOOST_HAS_PTHREADS	Platform	The platform support POSIX style threads.
BOOST_HAS_SCHED_YIELD	Platform	The platform has the POSIX API sched_yield.

Macro	Section	Description
BOOST_HAS_SGI_TYPE_TRAITS	Compiler, Standard library	The compiler has native support for SGI style type traits.
BOOST_HAS_STDINT_H	Platform	The platform has a <stdint.h>
BOOST_HAS_SLIST	Standard library	The C++ implementation provides the (SGI) slist class. When defined, BOOST_SLIST_HEADER will contain the name of the header needed to access slist and BOOST_STD_EXTENSION_NAMESPACE will provide the namespace in which slist resides.
BOOST_HAS_STLP_USE_FACET	Standard library	The standard library lacks a conforming std::use_facet, but has a workaround class-version that does the job. This is primarily for the STLport std lib.
BOOST_HAS_TR1_ARRAY	Standard library	The library has a TR1 conforming version of <array>.
BOOST_HAS_TR1_COMPLEX_OVERLOADS	Standard library	The library has a version of <complex> that supports passing scalars to the complex number algorithms.
BOOST_HAS_TR1_COMPLEX_INVERSE_TRIG	Standard library	The library has a version of <complex> that includes the new inverse trig functions from TR1.
BOOST_HAS_TR1_REFERENCE_WRAPPER	Standard library	The library has TR1 conforming reference wrappers in <functional>.
BOOST_HAS_TR1_RESULT_OF	Standard library	The library has a TR1 conforming result_of template in <functional>.
BOOST_HAS_TR1_MEM_FN	Standard library	The library has a TR1 conforming mem_fn function template in <functional>.
BOOST_HAS_TR1_BIND	Standard library	The library has a TR1 conforming bind function template in <functional>.
BOOST_HAS_TR1_FUNCTION	Standard library	The library has a TR1 conforming function class template in <functional>.
BOOST_HAS_TR1_HASH	Standard library	The library has a TR1 conforming hash function template in <functional>.
BOOST_HAS_TR1_SHARED_PTR	Standard library	The library has a TR1 conforming shared_ptr class template in <memory>.
BOOST_HAS_TR1_RANDOM	Standard library	The library has a TR1 conforming version of <random>.
BOOST_HAS_TR1_REGEX	Standard library	The library has a TR1 conforming version of <regex>.
BOOST_HAS_TR1_TUPLE	Standard library	The library has a TR1 conforming version of <tuple>.
BOOST_HAS_TR1_TYPE_TRAITS	Standard library	The library has a TR1 conforming version of <type_traits>.
BOOST_HAS_TR1_UTILITY	Standard library	The library has the TR1 additions to <utility> (tuple interface to std::pair).

Macro	Section	Description
BOOST_HAS_TR1_UNORDERED_MAP	Standard library	The library has a TR1 conforming version of <unordered_map>.
BOOST_HAS_TR1_UNORDERED_SET	Standard library	The library has a TR1 conforming version of <unordered_set>.
BOOST_HAS_TR1	Standard library	Implies all the other BOOST_HAS_TR1_* macros should be set.
BOOST_HAS_THREADS	Platform, Compiler	Defined if the compiler, in its current translation mode, supports multiple threads of execution.
BOOST_HAS_TWO_ARG_USE_FACET	Standard library	The standard library lacks a conforming std::use_facet, but has a two argument version that does the job. This is primarily for the Rogue Wave std lib.
BOOST_HAS_UNISTD_H	Platform	The Platform provides <unistd.h>.
BOOST_HAS_WINTHREADS	Platform	The platform supports MS Windows style threads.
BOOST_MSVC_STD_ITERATOR	Standard library	Microsoft's broken version of std::iterator is being used. This implies that std::iterator takes no more than two template parameters.
BOOST_MSVC6_MEMBER_TEMPLATES	Compiler	Microsoft Visual C++ 6.0 has enough member template idiosyncrasies (being polite) that BOOST_NO_MEMBER_TEMPLATES is defined for this compiler. BOOST_MSVC6_MEMBER_TEMPLATES is defined to allow compiler specific workarounds. This macro gets defined automatically if BOOST_NO_MEMBER_TEMPLATES is not defined - in other words this is treated as a strict subset of the features required by the standard.
BOOST_HAS_STDINT_H	Platform	There are no 1998 C++ Standard headers <stdint.h> or <cstdint>, although the 1999 C Standard does include <stdint.h>. If <stdint.h> is present, <boost/stdint.h> can make good use of it, so a flag is supplied (signalling presence; thus the default is not present, conforming to the current C++ standard).

Macros that describe possible C++0x features

The following macros describe features that are likely to be included in the upcoming ISO C++ standard, C++0x, but have not yet been approved for inclusion in the language.

Macro	Description
BOOST_HAS_CONCEPTS	The compiler supports concepts.

Macros that describe C++0x features not supported

The following macros describe features in the upcoming ISO C++ standard, C++0x, that are not yet supported by a particular compiler.

Macro	Description
BOOST_NO_CHAR16_T	The compiler does not support type char16_t.
BOOST_NO_CHAR32_T	The compiler does not support type char32_t.
BOOST_NO_CONSTEXPR	The compiler does not support constexpr.
BOOST_NO_DECLTYPE	The compiler does not support decltype.
BOOST_NO_DEFAULTED_FUNCTIONS	The compiler does not support defaulted (= default) functions.
BOOST_NO_DELETED_FUNCTIONS	The compiler does not support deleted (= delete) functions.
BOOST_NO_EXPLICIT_CONVERSION_OPERATIONS	The compiler does not support explicit conversion operators (explicit operator T()).
BOOST_NO_EXTERN_TEMPLATE	The compiler does not support explicit instantiation declarations for templates (explicit template).
BOOST_NO_INITIALIZER_LISTS	The C++ compiler does not support C++0x initializer lists.
BOOST_NO_LONG_LONG	The compiler does not support long long.
BOOST_NO_RAW_LITERALS	The compiler does not support raw string literals.
BOOST_NO_RVALUE_REFERENCES	The compiler does not support r-value references.
BOOST_NO_SCOPED_ENUMS	The compiler does not support scoped enumerations (enum class).
BOOST_NO_STATIC_ASSERT	The compiler does not support static_assert.
BOOST_NO_STD_UNORDERD	The standard library does not support <unordered_map> and <unordered_set>.
BOOST_NO_UNICODE_LITERALS	The compiler does not support Unicode (u8, u, U) literals.
BOOST_NO_VARIADIC_TEMPLATES	The compiler does not support variadic templates.

Boost Helper Macros

The following macros are either simple helpers, or macros that provide workarounds for compiler/standard library defects.

Macro	Description
`BOOST_DEDUCED_TYPENAME`	Some compilers don't support the use of typename for dependent types in deduced contexts. This macro expands to nothing on those compilers, and typename elsewhere. For example, replace: `template <class T> void f(T, typename T::type);` with: `template <class T> void f(T, BOOST_DEDUCED_TYPENAME T::type);`
`BOOST_HASH_MAP_HEADER`	The header to include to get the SGI `hash_map` class. This macro is only available if `BOOST_HAS_HASH` is defined.
`BOOST_HASH_SET_HEADER`	The header to include to get the SGI `hash_set` class. This macro is only available if `BOOST_HAS_HASH` is defined.
`BOOST_SLIST_HEADER`	The header to include to get the SGI `slist` class. This macro is only available if `BOOST_HAS_SLIST` is defined.
`BOOST_STD_EXTENSION_NAMESPACE`	The namespace used for std library extensions (hashtable classes etc).

`BOOST_STATIC_CONSTANT(Type, assignment)`	On compilers which don't allow in-class initialization of static integral constant members, we must use enums as a workaround if we want the constants to be available at compile-time. This macro gives us a convenient way to declare such constants. For example instead of:
	`struct foo{ static const int value = 2; };`
	use:
	`struct foo{ BOOST_STATIC_CONSTANT(int, value = 2); };`

`BOOST_UNREACHABLE_RETURN(result)`	Normally evaluates to nothing, but evaluates to return x; if the compiler requires a return, even when it can never be reached.

<table>
<tr><th>Macro</th><th>Description</th></tr>
<tr><td rowspan="5">BOOST_EXPLICIT_TEMPLATE_TYPE(t) BOOST_EXPLICIT_TEMPLATE_NON_TYPE(t,v) BOOST_APPEND_EXPLICIT_TEMPLATE_TYPE(t) BOOST_APPEND_EXPLICIT_TEMPLATE_NON_TYPE(t,v)</td><td>Some compilers silently "fold" different function template instantiations if some of the template parameters don't appear in the function parameter list. For instance:</td></tr>
<tr><td><pre>#include <iostream>
#include <ostream>
#include <typeinfo>

template <int n>
void f() { std::cout << n << ' '; }

template <typename T>
void g() { std::cout << typeid(T).name() << ' '; }

int main() {
  f<1>();
  f<2>();

  g<int>();
  g<double>();
}</pre></td></tr>
<tr><td>incorrectly outputs <code>2 2 double double</code> on VC++ 6. These macros, to be used in the function parameter list, fix the problem without effects on the calling syntax. For instance, in the case above write:</td></tr>
<tr><td><pre>template <int n>
void f(BOOST_EXPLICIT_TEMPLATE_NON_TYPE(int, n)) { ... }

template <typename T>
void g(BOOST_EXPLICIT_TEMPLATE_TYPE(T)) { ... }</pre></td></tr>
<tr><td>Beware that they can declare (for affected compilers) a dummy defaulted parameter, so they

a) should be always invoked at the end of the parameter list

b) can't be used if your function template is multiply declared.

Furthermore, in order to add any needed comma separator, an <code>APPEND_*</code> version must be used when the macro invocation appears after a normal parameter declaration or after the invocation of another macro of this same group.</td></tr>
<tr><td>BOOST_USE_FACET(Type, loc)</td><td>When the standard library does not have a comforming <code>std::use_facet</code> there are various workarounds available, but they differ from library to library. This macro provides a consistent way to access a locale's facets. For example, replace: <code>std::use_facet<Type>(loc);</code> with: <code>BOOST_USE_FACET(Type, loc);</code> Note do not add a <code>std::</code> prefix to the front of <code>BOOST_USE_FACET</code>.</td></tr>
<tr><td>BOOST_HAS_FACET(Type, loc)</td><td>When the standard library does not have a comforming <code>std::has_facet</code> there are various workarounds available, but they differ from library to library. This macro provides a consistent way to check a locale's facets. For example, replace: <code>std::has_facet<Type>(loc);</code> with: <code>BOOST_HAS_FACET(Type, loc);</code> Note do not add a <code>std::</code> prefix to the front of <code>BOOST_HAS_FACET</code>.</td></tr>
</table>

Macro	Description
BOOST_NESTED_TEMPLATE	Member templates are supported by some compilers even though they can't use the A::template member<U> syntax, as a workaround replace: typedef typename A::template rebind<U> binder; with: typedef typename A::BOOST_NESTED_TEMPLATE rebind<U> binder;
BOOST_STRINGIZE(X)	Converts the parameter X to a string after macro replacement on X has been performed.
BOOST_JOIN(X,Y)	This piece of macro magic joins the two arguments together, even when one of the arguments is itself a macro (see 16.3.1 in C++ standard). This is normally used to create a mangled name in combination with a predefined macro such a __LINE__.

Boost Informational Macros

The following macros describe boost features; these are, generally speaking the only boost macros that should be tested in user code.

Macro	Header	Description
BOOST_VERSION	<boost/version.hpp>	Describes the boost version number in XXYYZZ format such that: (BOOST_VERSION % 100) is the sub-minor version, ((BOOST_VERSION / 100) % 1000) is the minor version, and (BOOST_VERSION / 100000) is the major version.
BOOST_NO_INT64_T	<boost/cstdint.hpp> <boost/stdint.h>	Defined if there are no 64-bit integral types: int64_t, uint64_t etc.
BOOST_NO_INTEGRAL_INT64_T	<boost/cstdint.hpp> <boost/stdint.h>	Defined if int64_t as defined by <boost/cstdint.hpp> is not usable in integral constant expressions.
BOOST_MSVC	<boost/config.hpp>	Defined if the compiler is really Microsoft Visual C++, as opposed to one of the many other compilers that also define _MSC_VER.
BOOST_INTEL	<boost/config.hpp>	Defined if the compiler is an Intel compiler, takes the same value as the compiler version macro.
BOOST_WINDOWS	<boost/config.hpp>	Defined if the Windows platfrom API is available.
BOOST_DINKUMWARE_STDLIB	<boost/config.hpp>	Defined if the dinkumware standard library is in use, takes the same value as the Dinkumware library version macro _CPPLIB_VER if defined, otherwise 1.
BOOST_NO_WREGEX	<boost/regex.hpp>	Defined if the regex library does not support wide character regular expressions.
BOOST_COMPILER	<boost/config.hpp>	Defined as a string describing the name and version number of the compiler in use. Mainly for debugging the configuration.
BOOST_STDLIB	<boost/config.hpp>	Defined as a string describing the name and version number of the standard library in use. Mainly for debugging the configuration.
BOOST_PLATFORM	<boost/config.hpp>	Defined as a string describing the name of the platform. Mainly for debugging the configuration.

Macros for libraries with separate source code

The following macros and helper headers are of use to authors whose libraries include separate source code, and are intended to address two issues: fixing the ABI of the compiled library, and selecting which compiled library to link against based upon the compilers settings.

ABI Fixing

When linking against a pre-compiled library it vital that the ABI used by the compiler when building the library *matches exactly* the ABI used by the code using the library. In this case ABI means things like the struct packing arrangement used, the name mangling scheme used, or the size of some types (enum types for example). This is separate from things like threading support, or runtime library variations, which have to be dealt with by build variants. To put this in perspective there is one compiler (Borland's) that has so many compiler options that make subtle changes to the ABI, that at least in theory there 3200 combinations, and that's without considering runtime library variations. Fortunately these variations can be managed by `#pragma`'s that tell the compiler what ABI to use for the types declared in your library. In order to avoid sprinkling `#pragma`'s all over the boost headers, there are some prefix and suffix headers that do the job. Typical usage is:

my_library.hpp

```
#ifndef MY_INCLUDE_GUARD
#define MY_INCLUDE_GUARD

// all includes go here:
#include <boost/config.hpp>
#include <whatever>

#include <boost/config/abi_prefix.hpp> // must be the last #include

namespace boost {

// your code goes here

}

#include <boost/config/abi_suffix.hpp> // pops abi_prefix.hpp pragmas

#endif // include guard
```

my_library.cpp

```
...
// nothing special need be done in the implementation file
...
```

The user can disable this mechanism by defining `BOOST_DISABLE_ABI_HEADERS`, or they can define `BOOST_ABI_PREFIX` and/or `BOOST_ABI_SUFFIX` to point to their own prefix/suffix headers if they so wish.

Automatic library selection

It is essential that users link to a build of a library which was built against the same runtime library that their application will be built against -if this does not happen then the library will not be binary compatible with their own code- and there is a high likelihood that their application will experience runtime crashes. These kinds of problems can be extremely time consuming and difficult to debug, and often lead to frustrated users and authors alike (simply selecting the right library to link against is not as easy as it seems when their are 6-8 of them to chose from, and some users seem to be blissfully unaware that there even are different runtimes available to them).

To solve this issue, some compilers allow source code to contain `#pragma`'s that instruct the linker which library to link against, all the user need do is include the headers they need, place the compiled libraries in their library search path, and the compiler and linker

do the rest. Boost.config supports this via the header <boost/config/auto_link.hpp>, before including this header one or more of the following macros need to be defined:

BOOST_LIB_NAME	Required: An identifier containing the basename of the library, for example 'boost_regex'.
BOOST_DYN_LINK	Optional: when set link to dll rather than static library.
BOOST_LIB_DIAGNOSTIC	Optional: when set the header will print out the name of the library selected (useful for debugging).

If the compiler supports this mechanism, then it will be told to link against the appropriately named library, the actual algorithm used to mangle the name of the library is documented inside <boost/config/auto_link.hpp> and has to match that used to create the libraries via bjam 's install rules.

my_library.hpp

```
...
//
// Don't include auto-linking code if the user has disabled it by
// defining BOOST_ALL_NO_LIB, or BOOST_MY_LIBRARY_NO_LIB, or if this
// is one of our own source files (signified by BOOST_MY_LIBRARY_SOURCE):
//
#if !defined(BOOST_ALL_NO_LIB) && !defined(BOOST_MY_LIBRARY_NO_LIB) && !defined(BOOST_MY_LIB↵
RARY_SOURCE)
#  define BOOST_LIB_NAME boost_my_library
#  ifdef BOOST_MY_LIBRARY_DYN_LINK
#     define BOOST_DYN_LINK
#  endif
#  include <boost/config/auto_link.hpp>
#endif
...
```

my_library.cpp

```
// define BOOST_MY_LIBRARY_SOURCE so that the header knows that the
// library is being built (possibly exporting rather than importing code)
//
#define BOOST_MY_LIBRARY_SOURCE

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

Guidelines for Boost Authors

The <boost/config.hpp> header is used to pass configuration information to other boost files, allowing them to cope with platform dependencies such as arithmetic byte ordering, compiler pragmas, or compiler shortcomings. Without such configuration information, many current compilers would not work with the Boost libraries.

Centralizing configuration information in this header reduces the number of files that must be modified when porting libraries to new platforms, or when compilers are updated. Ideally, no other files would have to be modified when porting to a new platform.

Configuration headers are controversial because some view them as condoning broken compilers and encouraging non-standard subsets. Adding settings for additional platforms and maintaining existing settings can also be a problem. In other words, configuration headers are a necessary evil rather than a desirable feature. The boost config.hpp policy is designed to minimize the problems and maximize the benefits of a configuration header.

Note that:

- Boost library implementers are not required to "#include <boost/config.hpp>", and are not required in any way to support compilers that do not comply with the C++ Standard (ISO/IEC 14882).

- If a library implementer wishes to support some non-conforming compiler, or to support some platform specific feature, "`#include <boost/config.hpp>`" is the preferred way to obtain configuration information not available from the standard headers such as `<climits>`, etc.
- If configuration information can be deduced from standard headers such as `<climits>`, use those standard headers rather than `<boost/config.hpp>`.
- Boost files that use macros defined in `<boost/config.hpp>` should have sensible, standard conforming, default behavior if the macro is not defined. This means that the starting point for porting `<boost/config.hpp>` to a new platform is simply to define nothing at all specific to that platform. In the rare case where there is no sensible default behavior, an #error message should describe the problem.
- If a Boost library implementer wants something added to `config.hpp`, post a request on the Boost mailing list. There is no guarantee such a request will be honored; the intent is to limit the complexity of config.hpp.
- The intent is to support only compilers which appear on their way to becoming C++ Standard compliant, and only recent releases of those compilers at that.
- The intent is not to disable mainstream features now well-supported by the majority of compilers, such as namespaces, exceptions, RTTI, or templates.

Disabling Compiler Warnings

The header `<boost/config/warning_disable.hpp>` can be used to disable certain compiler warings that are hard or impossible to otherwise remove.

Note that:

- This header **should never be included by another Boost header**, it should only ever be used by a library source file or a test case.
- The header should be included **before you include any other header**.
- This header only disables warnings that are hard or impossible to otherwise deal with, and which are typically emitted by one compiler only, or in one compilers own standard library headers.

Currently it disables the following warnings:

Compiler	Warning
Visual C++ 8 and later	C4996: Error 'function': was declared deprecated
Intel C++	Warning 1786: relates to the use of "deprecated" standard library functions rather like C4996 in Visual C++.

Adding New Defect Macros

When you need to add a new defect macro - either to fix a problem with an existing library, or when adding a new library - distil the issue down to a simple test case; often, at this point other (possibly better) workarounds may become apparent. Secondly always post the test case code to the boost mailing list and invite comments; remember that C++ is complex and that sometimes what may appear a defect, may in fact turn out to be a problem with the authors understanding of the standard.

When you name the macro, follow the `BOOST_NO_`*SOMETHING* naming convention, so that it's obvious that this is a macro reporting a defect.

Finally, add the test program to the regression tests. You will need to place the test case in a `.ipp` file with the following comments near the top:

```
//  MACRO:         BOOST_NO_FOO
//  TITLE:         foo
//  DESCRIPTION:   If the compiler fails to support foo
```

These comments are processed by the autoconf script, so make sure the format follows the one given. The file should be named "`boost_no_foo.ipp`", where foo is the defect description - try and keep the file name under the Mac 30 character filename limit though. You will also need to provide a function prototype "`int test()`" that is declared in a namespace with the same name as the macro, but in all lower case, and which returns zero on success:

```
namespace boost_no_foo {

int test()
{
    // test code goes here:
    //
    return 0;
}

}
```

Once the test code is in place in libs/config/test, updating the configuration test system proceeds as:

- cd into `libs/config/tools` and run `bjam` : this generates the `.cpp` file test cases from the `.ipp` file, updates the libs/config/test/all/Jamfile.v2, `config_test.cpp` and `config_info.cpp`.
- cd into `libs/config/test/all` and run `bjam` *MACRONAME* *`compiler-list`* : where *MACRONAME* is the name of the new macro, and *`compiler-list`* is a space separated list of compilers to test with. You should see the tests pass with those compilers that don't have the defect, and fail with those that do.
- cd into `libs/config/test` and run `bjam config_info config_test` *`compiler-list`* : `config_info` should build and run cleanly for all the compilers in *`compiler-list`* while `config_test` should fail for those that have the defect, and pass for those that do not.

Then you should:

- Define the defect macro in those config headers that require it.
- Document the macro in this documentation (please do not forget this step!!)
- Commit everything.
- Keep an eye on the regression tests for new failures in Boost.Config caused by the addition.
- Start using the macro.

Adding New Feature Test Macros

When you need to add a macro that describes a feature that the standard does not require, follow the convention for adding a new defect macro (above), but call the macro `BOOST_HAS_FOO`, and name the test file "`boost_has_foo.ipp`". Try not to add feature test macros unnecessarily, if there is a platform specific macro that can already be used (for example `_WIN32`, `__BEOS__`, or `__linux`) to identify the feature then use that. Try to keep the macro to a feature group, or header name, rather than one specific API (for example `BOOST_HAS_NL_TYPES_H` rather than `BOOST_HAS_CATOPEN`). If the macro describes a POSIX feature group, then add boilerplate code to <boost/config/suffix.hpp> to auto-detect the feature where possible (if you are wondering why we can't use POSIX feature test macro directly, remember that many of these features can be added by third party libraries, and are not therefore identified inside `<unistd.h>`).

Modifying the Boost Configuration Headers

The aim of boost's configuration setup is that the configuration headers should be relatively stable - a boost user should not have to recompile their code just because the configuration for some compiler that they're not interested in has changed. Separating the configuration into separate compiler/standard library/platform sections provides for part of this stability, but boost authors require some amount of restraint as well, in particular:

<boost/config.hpp> should never change, don't alter this file.

<boost/config/user.hpp> is included by default, don't add extra code to this file unless you have to. If you do, please remember to update libs/config/tools/configure.in as well.

<boost/config/suffix.hpp> is always included so be careful about modifying this file as it breaks dependencies for everyone. This file should include only "boilerplate" configuration code, and generally should change only when new macros are added.

<boost/config/select_compiler_config.hpp>, <boost/config/select_platform_config.hpp> and <boost/config/select_stdlib_config.hpp> are included by default and should change only if support for a new compiler/standard library/platform is added.

The compiler/platform/standard library selection code is set up so that unknown platforms are ignored and assumed to be fully standards compliant - this gives unknown platforms a "sporting chance" of working "as is" even without running the configure script.

When adding or modifying the individual mini-configs, assume that future, as yet unreleased versions of compilers, have all the defects of the current version. Although this is perhaps unnecessarily pessimistic, it cuts down on the maintenance of these files, and experience suggests that pessimism is better placed than optimism here!

Rationale

The problem with many traditional "textbook" implementations of configuration headers (where all the configuration options are in a single "monolithic" header) is that they violate certain fundamental software engineering principles which would have the effect of making boost more fragile, more difficult to maintain and more difficult to use safely. You can find a description of the principles from the following article.

The problem

Consider a situation in which you are concurrently developing on multiple platforms. Then consider adding a new platform or changing the platform definitions of an existing platform. What happens? Everything, and this does literally mean everything, recompiles. Isn't it quite absurd that adding a new platform, which has absolutely nothing to do with previously existing platforms, means that all code on all existing platforms needs to be recompiled?

Effectively, there is an imposed physical dependency between platforms that have nothing to do with each other. Essentially, the traditional solution employed by configuration headers does not conform to the Open-Closed Principle:

> **"A module should be open for extension but closed for modification."**

Extending a traditional configuration header implies modifying existing code.

Furthermore, consider the complexity and fragility of the platform detection code. What if a simple change breaks the detection on some minor platform? What if someone accidentally or on purpose (as a workaround for some other problem) defines some platform dependent macros that are used by the detection code? A traditional configuration header is one of the most volatile headers of the entire library, and more stable elements of Boost would depend on it. This violates the Stable Dependencies Principle:

> **"Depend in the direction of stability."**

After even a minor change to a traditional configuration header on one minor platform, almost everything on every platform should be tested if we follow sound software engineering practice.

Another important issue is that it is not always possible to submit changes to `<boost/config.hpp>`. Some boost users are currently working on platforms using tools and libraries that are under strict Non-Disclosure Agreements. In this situation it is impossible to

submit changes to a traditional monolithic configuration header, instead some method by which the user can insert their own configuration code must be provided.

The solution

The approach taken by boost's configuration headers is to separate configuration into three orthogonal parts: the compiler, the standard library and the platform. Each compiler/standard library/platform gets its own mini-configuration header, so that changes to one compiler's configuration (for example) does not affect other compilers. In addition there are measures that can be taken both to omit the compiler/standard library/platform detection code (so that adding support to a new platform does not break dependencies), or to freeze the configuration completely; providing almost complete protection against dependency changes.

Acknowledgements

Beman Dawes provided the original `config.hpp` and part of this document.

Vesa Karvonen provided a description of the principles (see rationale) and put together an early version of the current configuration setup.

John Maddock put together the configuration current code, the test programs, the configuration script and the reference section of this document.

Matias Capeletto converted the docs to quickbook format.

Numerous boost members, past and present, have contributed fixes to boost's configuration.