Boost.Config

Vesa Karvonen, John Maddock Beman Dawes

Copyright © 2001-2007 Beman Dawes, Vesa Karvonen, John Maddock

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 Trac 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:

2

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 svn 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.

Macro	Description
`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".
`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.Thread 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.Thread 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.

Macro	Description
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_COM-PILER_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 compiled, 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 is 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 C++03 defects

The following macros all describe features that are required by the C++03 standard, if one of the following macros is defined, then it represents a defect in the compiler's conformance with the 2003 standard.

Macro	Section	Description
`BOOST_BCB_PARTIAL_SPECIALIZA-TION_BUG`	Compiler	The compiler exhibits certain partial specialisation bug - probably Borland C++ Builder specific.
`BOOST_FUNCTION_SCOPE_USING_DE-CLARATION_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_DEPEND-ENT_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_COMPLETE_VALUE_INITIAL-IZATION`	Compiler	Compiler has not completely implemented value-initialization. See also The Utility/Value Init docs
`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_SPECIALIZA-TIONS`	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_FENV_H`	Platform, Standard library	The C standard library doesn't provide `<fenv.h>`. `<boost/detail/fenv.hpp>` should be included instead of `<fenv.h>` for maximum portability on platforms which do provide `<fenv.h>`.

Macro	Section	Description

BOOST_NO_DEPENDENT_NESTED_DERIV-ATIONS	Compiler	The compiler fails to compile a nested class that has a dependent base class:
		
		```cpp\ntemplate<typename T>\nstruct foo : {\n template<typename U>\n struct bar : public U {};\n```
		
		};

BOOST_NO_DEPENDENT_TYPES_IN_TEM-PLATE_VALUE_PARAMETERS	Compiler	Template value parameters cannot have a dependent type, for example: ```cpp\ntemplate<class T, type↵\nname T::type value>\nclass X { ... };\n```
BOOST_NO_EXCEP-TION_STD_NAMESPACE	Standard Library	The standard library does not put some or all of the contents of `<exception>` in namespace std.
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 require-ment 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_FUNCTION_TEMPLATE_OR-DERING	Compiler	The compiler does not perform function template ordering or its function template ordering is incorrect. ```cpp\n// #1\ntemplate<class T> void f(T);\n\n// #2\ntem↵\nplate<class T,class U> void f(T(*)(U));\n\nvoid bar(int);\n\nf(&bar); // should choose #2.\n```
BOOST_NO_INCLASS_MEMBER_INITIAL-IZATION	Compiler	Compiler violates std::9.4.2/4.

Macro	Section	Description
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_CXX11_NUMERIC_LIMITS	Standard library	C++11 additions to `std::numeric_limits` are not available for use. `static function numeric_limits<T>::lowest()` the lowest finite value representable by the numeric type. `static int const max_digits10` the number of decimal digits that are required to make sure that two distinct values of the type have distinct decimal representations. `template<> class numeric_limits<char16_t>;`, see also BOOST_NO_CXX11_CHAR16_T, `template<> class numeric_limits<char32_t>;` see also BOOST_NO_CXX11_CHAR32_T. Replaces BOOST_NO_NUMERIC_LIMITS_LOWEST.
BOOST_NO_LIMITS_COMPILE_TIME_CONSTANTS	Standard library	Constants such as `numeric_limits<T>::is_signed` are not available for use at compile-time.

Macro	Section	Description
BOOST_NO_LONG_LONG_NUMERIC_LIM-ITS	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.
BOOST_NO_MEMBER_FUNCTION_SPE-CIALIZATIONS	Compiler	The compiler does not support the specialization of individual member functions of template classes.
BOOST_NO_MEMBER_TEM-PLATE_KEYWORD	Compiler	If the compiler supports member templates, but not the template keyword when accessing member template classes.
BOOST_NO_MEMBER_TEM-PLATE_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_LIM-ITS	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_SPECIALIZA-TION_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_MEM-BER_CONST	Compiler	The compiler does not correctly handle pointers to const member functions, preventing use of these in overloaded function templates. See `<boost/function-al.hpp>` for example.
BOOST_NO_POINTER_TO_MEMBER_TEM-PLATE_PARAMETERS	Compiler	Pointers to members don't work when used as template parameters.

Macro	Section	Description
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_SFINAE_EXPR	Compiler	The compiler does not support usage of SFINAE with arbitrary expressions.
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.
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_AS-SIGN	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.

Macro	Section	Description
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_SPE-CIALIZATION	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_CON-STRUCTORS	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_DE-TECTION	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_OVER-LOADS_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_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 type FTIME.
`BOOST_HAS_GETSYSTEMTIMEASFILE-TIME`	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_INT128`	Compiler	The compiler has `__int128` and `unsigned __int128` as native types which are distinct from all the regular C++ integer types.
`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>`.

Macro	Section	Description
`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_PRAGMA_ONCE`	Compiler	The compiler recognizes the `#pragma once` directive which tells that the containing header should be included only once while preprocessing the current translation unit. The pragma may improve compile times of large projects with some compilers.
`BOOST_HAS_PTHREAD_DELAY_NP`	Platform	The platform has the POSIX API `pthread_delay_np`.
`BOOST_HAS_PTHREAD_MUTEXATTR_SET-TYPE`	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`.
`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.

Macro	Section	Description
`BOOST_HAS_TR1_ARRAY`	Standard library	The library has a TR1 conforming version of `<array>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_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_IN-VERSE_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>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_FUNCTION-AL.
`BOOST_HAS_TR1_RESULT_OF`	Standard library	The library has a TR1 conforming result_of template in `<functional>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_FUNCTION-AL.
`BOOST_HAS_TR1_MEM_FN`	Standard library	The library has a TR1 conforming mem_fn function template in `<function-al>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_FUNCTION-AL.
`BOOST_HAS_TR1_BIND`	Standard library	The library has a TR1 conforming bind function template in `<functional>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_FUNCTION-AL.

Macro	Section	Description
BOOST_HAS_TR1_FUNCTION	Standard library	The library has a TR1 conforming function class template in `<functional>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_FUNCTIONAL.
BOOST_HAS_TR1_HASH	Standard library	The library has a TR1 conforming hash function template in `<functional>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_FUNCTIONAL.
BOOST_HAS_TR1_SHARED_PTR	Standard library	The library has a TR1 conforming `shared_ptr` class template in `<memory>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_SMART_PTR.
BOOST_HAS_TR1_RANDOM	Standard library	The library has a TR1 conforming version of `<random>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_RANDOM.
BOOST_HAS_TR1_REGEX	Standard library	The library has a TR1 conforming version of `<regex>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_REGEX.
BOOST_HAS_TR1_TUPLE	Standard library	The library has a TR1 conforming version of `<tuple>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_TUPLE.
BOOST_HAS_TR1_TYPE_TRAITS	Standard library	The library has a TR1 conforming version of `<type_traits>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_TYPE_TRAITS.

Macro	Section	Description
`BOOST_HAS_TR1_UTILITY`	Standard library	The library has the TR1 additions to `<utility>` (tuple interface to `std::pair`). This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_TUPLE.
`BOOST_HAS_TR1_UNORDERED_MAP`	Standard library	The library has a TR1 conforming version of `<unordered_map>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_UN-ORDERED_MAP.
`BOOST_HAS_TR1_UNORDERED_SET`	Standard library	The library has a TR1 conforming version of `<unordered_set>`. This macro is only guaranteed to be defined after including one of the headers from Boost.TR1. Further this macro is now deprecated in favour of BOOST_NO_CXX11_HDR_UN-ORDERED_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.

Macro	Section	Description
`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 work-arounds. 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++ future features

The following macros describe features that may be included in some future ISO C++ standard, but have not yet been approved for inclusion in the language.

Macro	Description
`BOOST_HAS_CONCEPTS`	The compiler supports concepts.

Macros that describe C++11 features not supported

The following macros describe features in the 2011 ISO C++ standard, formerly known as C++0x, that are not yet supported by a particular compiler or library.

Macro	Description
BOOST_NO_CXX11_ALIGNAS	The compiler does not support the `alignas` keyword.
BOOST_NO_CXX11_ALLOCATOR	The standard library does not provide a C++11 version of `std::allocator` in <memory>.
BOOST_NO_CXX11_ATOMIC_SP	The standard library <memory> does not support atomic smart pointer operations.
BOOST_NO_CXX11_HDR_ARRAY	The standard library does not provide header <array>.
BOOST_NO_CXX11_HDR_CHRONO	The standard library does not provide header <chrono>.
BOOST_NO_CXX11_HDR_CODECVT	The standard library does not provide header <codecvt>.
BOOST_NO_CXX11_HDR_CONDITION_VARIABLE	The standard library does not provide header <condition_variable>.
BOOST_NO_CXX11_HDR_FORWARD_LIST	The standard library does not provide header <forward_list>.
BOOST_NO_CXX11_HDR_FUNCTIONAL	The standard library does not provide a C++11 compatible version of <functional>.
BOOST_NO_CXX11_HDR_FUTURE	The standard library does not provide header <future>.
BOOST_NO_CXX11_HDR_INITIALIZER_LIST	The standard library does not provide header <initializer_list>.
BOOST_NO_CXX11_HDR_MUTEX	The standard library does not provide header <mutex>.
BOOST_NO_CXX11_HDR_RANDOM	The standard library does not provide header <random>.
BOOST_NO_CXX11_HDR_RATIO	The standard library does not provide header <ratio>.
BOOST_NO_CXX11_HDR_REGEX	The standard library does not provide header <regex>.
BOOST_NO_CXX11_HDR_SYSTEM_ERROR	The standard library does not provide header <system_error>.
BOOST_NO_CXX11_HDR_THREAD	The standard library does not provide header <thread>.
BOOST_NO_CXX11_HDR_TUPLE	The standard library does not provide header <tuple>.
BOOST_NO_CXX11_HDR_TYPEINDEX	The standard library does not provide header <typeindex>.
BOOST_NO_CXX11_HDR_TYPE_TRAITS	The standard library does not provide header <type_traits>.
BOOST_NO_CXX11_HDR_UNORDERED_MAP	The standard library does not provide header <unordered_map>.
BOOST_NO_CXX11_HDR_UNORDERED_SET	The standard library does not provide header <unordered_set>.
BOOST_NO_CXX11_INLINE_NAMESPACES	The compiler does not support inline namespaces.
BOOST_NO_CXX11_SMART_PTR	The standard library header <memory> has no shared_ptr and unique_ptr.
BOOST_NO_CXX11_AUTO_DECLARATIONS	The compiler does not support type deduction for variables declared with the `auto` keyword (`auto var = ...;`).

Macro	Description
BOOST_NO_CXX11_AUTO_MULTIDECLARATIONS	The compiler does not support type deduction for multiple variables declared with the `auto` keyword (`auto var = ...,` `*ptr = ...;`).
BOOST_NO_CXX11_CHAR16_T	The compiler does not support type `char16_t`.
BOOST_NO_CXX11_CHAR32_T	The compiler does not support type `char32_t`.
BOOST_NO_CXX11_TEMPLATE_ALIASES	The compiler does not support template aliases.
BOOST_NO_CXX11_CONSTEXPR	The compiler does not support `constexpr`.
BOOST_NO_CXX11_DECLTYPE	The compiler does not support `decltype`.
BOOST_NO_CXX11_DECLTYPE_N3276	The compiler does not support the extension to `decltype` described in N3276, accepted in Madrid, March 2011.
BOOST_NO_CXX11_DEFAULTED_FUNCTIONS	The compiler does not support defaulted (`= default`) functions.
BOOST_NO_CXX11_DELETED_FUNCTIONS	The compiler does not support deleted (`= delete`) functions.
BOOST_NO_CXX11_EXPLICIT_CONVERSION_OPERATORS	The compiler does not support explicit conversion operators (`explicit operator T()`).
BOOST_NO_CXX11_EXTERN_TEMPLATE	The compiler does not support explicit instantiation forward declarations for templates (`extern template ...`).
BOOST_NO_CXX11_FUNCTION_TEMPLATE_DEFAULT_ARGS	The compiler does not support default template arguments for function templates.
BOOST_NO_CXX11_LAMBDAS	The compiler does not support Lambdas.
BOOST_NO_CXX11_LOCAL_CLASS_TEMPLATE_PARAMETERS	The compiler does not allow to pass local classes as template parameters (this macro intentionally does not control passing of unnamed types as template parameters, see also N2657).
BOOST_NO_LONG_LONG	The compiler does not support `long long`.
BOOST_NO_CXX11_NOEXCEPT	The compiler does not support `noexcept`.
BOOST_NO_CXX11_NULLPTR	The compiler does not support `nullptr`.
BOOST_NO_CXX11_RANGE_BASED_FOR	The compiler does not support range-based for statements.
BOOST_NO_CXX11_RAW_LITERALS	The compiler does not support raw string literals.
BOOST_NO_CXX11_RVALUE_REFERENCES	The compiler does not support r-value references.
BOOST_NO_CXX11_SCOPED_ENUMS	The compiler does not support scoped enumerations (`enum class`).
BOOST_NO_CXX11_STATIC_ASSERT	The compiler does not support `static_assert`.
BOOST_NO_CXX11_STD_UNORDERED	The standard library does not support <unordered_map> and <unordered_set>.

Macro	Description
`BOOST_NO_CXX11_TRAILING_RESULT_TYPES`	The compiler does not support the new function result type specification syntax (e.g. `auto foo(T) -> T;`).
`BOOST_NO_CXX11_UNICODE_LITERALS`	The compiler does not support Unicode (`u8`, `u`, `U`) literals.
`BOOST_NO_CXX11_UNIFIED_INITIALIZATION_SYNTAX`	The compiler does not support the C++11 Unified Initialization Syntax.
`BOOST_NO_CXX11_USER_DEFINED_LITERALS`	The compiler does not support user defined literals.
`BOOST_NO_CXX11_VARIADIC_TEMPLATES`	The compiler does not support variadic templates.
`BOOST_NO_CXX11_VARIADIC_MACROS`	The compiler does not support variadic macros.

Macros that allow use of C++11 features with C++03 compilers

The following macros allow use of C++11 features even with compilers that do not yet provide compliant C++11 support.

Macro	Description
`BOOST_ALIGNMENT(X)`, `BOOST_NO_ALIGNMENT`	Some compilers don't support the `alignas` keyword but provide other means to specify alignment (usually, through compiler-specific attributes). The macro `BOOST_ALIGNMENT(X)` will expand to the `alignas(X)` keyword if the compiler supports it or to some compiler-specific attribute to achieve the specified alignment. If no such compiler-specific attribute is known then `BOOST_ALIGNMENT(X)` will expand to nothing and `BOOST_NO_ALIGNMENT` will be defined. Unlike native `alignas`, `X` must always be a compile-time integer constant. The macro can be used to specify alignment of types and data: ```cpp\nstruct BOOST_ALIGNMENT(16) my_data\n{\n char c[16];\n};\nBOOST_ALIGNMENT(8) int arr[32];\n```

`BOOST_CONSTEXPR`	Some compilers don't support the use of `constexpr`. This macro expands to nothing on those compilers, and `constexpr` elsewhere. For example, when defining a constexpr function or constructor replace:
	
	 ```cpp\nconstexpr tuple();\n```
	
	 with:
	
	 ```cpp\nBOOST_CONSTEXPR tuple();\n```

`BOOST_CONSTEXPR_OR_CONST`	Some compilers don't support the use of `constexpr`. This macro expands to `const` on those compilers, and `constexpr` elsewhere. For example, when defining const expr variables replace:
	
	 ```cpp\nstatic constexpr UIntType xor_mask = a;\n```
	
	 with:
	
	 ```cpp\nstatic BOOST_CONSTEX↵\nPR_OR_CONST UIntType xor_mask = a;\n```

`BOOST_STATIC_CONSTEXPR`	This is a shortcut for `static BOOST_CONSTEXPR_OR_CONST`. For example, when defining const expr variables replace:
	
	 ```cpp\nstatic constexpr UIntType xor_mask = a;\n```
	
	 with:
	
	 ```cpp\nBOOST_STATIC_CONSTEXPR UIntType xor_mask = a;\n```

Macro	Description
| BOOST_DEFAULTED_FUNCTION(fun, body) | This macro is intended to be used within a class definition in order to declare a default implementation of function `fun`. For the compilers that do not support C++11 defaulted functions the macro will expand into an inline function definition with the `body` implementation. For example:<br><br>```cpp
struct my_struct
{
    BOOST_DEFAULTED_FUNCTION(my_struct(), {})
};
```<br><br>is equivalent to:<br><br>```cpp
struct my_struct
{
    my_struct() = default;
};
```<br><br>or:<br><br>```cpp
struct my_struct
{
    my_struct() {}
};
``` |

Macro	Description
BOOST_DELETED_FUNCTION(fun)	This macro is intended to be used within a class definition in order to declare a deleted function fun. For the compilers that do not support C++11 deleted functions the macro will expand into a private function declaration with no definition. Since the macro may change the access mode, it is recommended to use this macro at the end of the class definition. For example:

```cpp
struct noncopyable
{
    BOOST_DELETED_FUNCTION(noncopy↵
able const&))
    BOOST_DELETED_FUNCTION(noncopyable& oper↵
ator= (noncopyable const&))
};
```

is equivalent to:

```cpp
struct noncopyable
{
    noncopyable(noncopyable const&) = delete;
     noncopyable& operator= (noncopy↵
able const&) = delete;
};
```

or:

```cpp
struct noncopyable
{
private:
    noncopyable(noncopyable const&);
    noncopyable& operator= (noncopy↵
able const&);
};
```

Macro	Description

`BOOST_NOEXCEPT`
`BOOST_NOEXCEPT_OR_NOTHROW`
`BOOST_NOEXCEPT_IF(Predicate)`
`BOOST_NOEXCEPT_EXPR(Expression)`
	If `BOOST_NO_CXX11_NOEXCEPT` is defined (i.e. C++03 compliant compilers) these macros are defined as:
	
	```\n#define BOOST_NOEXCEPT\n#define BOOST_NOEXCEPT_OR_↵\nTHROW throw()\n#define BOOST_NOEXCEPT_IF(Predic↵\nate)\n#define BOOST_NOEXCEPT_EXPR(Ex↵\npression) false\n```
	
	If `BOOST_NO_CXX11_NOEXCEPT` is not defined (i.e. C++11 compliant compilers) they are defined as:
	
	```\n#define BOOST_NOEXCEPT noexcept\n#define BOOST_NOEXCEPT_OR_NO↵\nTHROW noexcept\n#define BOOST_NOEXCEPT_IF(Predic↵\nate) noexcept((Predicate))\n#define BOOST_NOEXCEPT_EXPR(Ex↵\npression) noexcept((Expression))\n```

`BOOST_MSVC_ENABLE_2012_NOV_CTP`	For Microsoft Visual C++ 2012, enable the C++11 features supplied by the November 2012 Community Technology Preview. These features are not automatically enabled because the CTP is non-supported alpha code that is not recommended for production use. This macro must be defined before including any Boost headers, and must be defined for all translation units in the program, including Boost library builds. This macro will no longer have any effect once an official Microsoft release supports the CTP features.

Boost Helper Macros

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

Macro	Description

`BOOST_WORKAROUND`	This macro is used where a compiler specific workaround is required that is not otherwise described by one of the other Boost.Config macros. To use the macro you must first
	
	```#include <boost/detail/workaround.hpp>```
	
	usage is then:
	
	```#if BOOST_WORKAROUND(MACRONAME, CONDITION) // workaround code goes here... #else // Standard conforming code goes here... #endif```
	
	where `MACRONAME` is a macro that usually describes the version number to be tested against, and `CONDITION` is a comparison operator followed by a value. For example `BOOST_WORK-AROUND(BOOST_INTEL, <= 1010)` would evaluate to `1` for Intel C++ 10.1 and earlier.
	
	The macro can also be used with `BOOST_TESTED_AT` if all current compiler versions exhibit the issue, but the issue is expected to be fixed at some later point.
	
	For example `BOOST_WORKAROUND(__BORLANDC__, BOOST_TESTED_AT(0x590))` would normally evaluate to `1` for all values of `__BORLANDC__` *unless* the macro `BOOST_DE-TECT_OUTDATED_WORKAROUNDS` is defined, in which case evaluates to (`__BORLANDC__ <= 0x590`).
	
	Note: the ultimate source of documentation for this macro is in boost/detail/workaround.hpp.

`BOOST_PREVENT_MACRO_SUBSTITUTION`	Sometimes you have a function name with the same name as a C macro, for example "min" and "max" member functions, in which case one can prevent the function being expanded as a macro using:
	
	```someclass.min BOOST_PREVENT_MACRO_SUBSTITU↵ TION(arg1, arg2);```
	
	The following also works in most, but not all, contexts:
	
	```(someclass.max)(arg1, arg2);```

`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_DE-DUCED_TYPENAME T::type);`

Macro	Description
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:
	
	```cpp\nstruct foo{\n static const int value = 2;\n};\n```
	
	use:
	
	```cpp\nstruct foo{\n BOOST_STATIC_CONSTANT(int, value = 2);\n};\n```

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.

Macro	Description
BOOST_FALLTHROUGH	The BOOST_FALLTHROUGH macro can be used to annotate implicit fall-through between switch labels:

```
switch (x) {
case 40:
case 41:
    if (truth_is_out_there) {
        ++x;
        BOOST_FALLTHROUGH;  // Use instead ↵
of/along with annotations in
        // comments.
    } else {
      return x;
    }
    case 42:
      ...
```

As shown in the example above, the BOOST_FALLTHROUGH macro should be followed by a semicolon. It is designed to mimic control-flow statements like 'break;', so it can be placed in most places where 'break;' can, but only if there are no statements on the execution path between it and the next switch label.

When compiled with Clang >3.2 in C++11 mode, the BOOST_FALLTHROUGH macro is expanded to `[[clang::fallthrough]]` attribute, which is analysed when performing switch labels fall-through diagnostic ('-Wimplicit-fallthrough'). See clang documentation on language extensions for details.

When used with unsupported compilers, the BOOST_FALLTHROUGH macro has no effect on diagnostics.

In either case this macro has no effect on runtime behavior and performance of code.

Macro	Description

`BOOST_EXPLICIT_TEMPLATE_TYPE(t)` `BOOST_EXPLI-CIT_TEMPLATE_NON_TYPE(t,v)` `BOOST_APPEND_EXPLI-CIT_TEMPLATE_TYPE(t)` `BOOST_APPEND_EXPLICIT_TEM-PLATE_NON_TYPE(t,v)`	Some compilers silently "fold" different function template instantiations if some of the template parameters don't appear in the function parameter list. For instance:
	
	```cpp\n#include <iostream>\n#include <ostream>\n#include <typeinfo>\n\ntemplate <int n>\nvoid f() { std::cout << n << ' '; }\n\ntemplate <typename T>\nvoid g() { std::cout << typeid(T).name() << '\n'; }\n\nint main() {\n f<1>();\n f<2>();\n\n g<int>();\n g<double>();\n}\n```
	
	incorrectly outputs `2 2 double double` 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:
	
	```cpp\ntemplate <int n>\nvoid f(BOOST_EXPLICIT_TEMPLATE_NON_TYPE(int, n)) { ... }\n\ntemplate <typename T>\nvoid g(BOOST_EXPLICIT_TEMPLATE_TYPE(T)) { ... }\n```
	
	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 `APPEND_*` 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.

`BOOST_USE_FACET(Type, loc)`	When the standard library does not have a conforming `std::use_facet` 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: `std::use_facet<Type>(loc);` with: `BOOST_USE_FACET(Type, loc);` Note do not add a `std::` prefix to the front of `BOOST_USE_FACET`.

Macro	Description
`BOOST_HAS_FACET(Type, loc)`	When the standard library does not have a comforming `std::has_facet` 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: `std::has_facet<Type>(loc);` with: `BOOST_HAS_FACET(Type, loc);` Note do not add a `std::` prefix to the front of `BOOST_HAS_FACET`.
`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_FORCEINLINE`	This macro can be used in place of the `inline` keyword to instruct the compiler that a function should always be inlined. Overuse of this macro can lead to significant bloat, while good use can increase performance in certain cases, such as computation-intensive code built through generative programming techniques. Usage example: ```cpp\ntemplate<class T>\nBOOST_FORCEINLINE T& f(T& t)\n{\n return t;\n}\n``` Note that use of this macro can lead to cryptic error messages with some compilers. Consider defining it to `inline` before including the Boost.Config header in order to be able to debug errors more easily.
`BOOST_NOINLINE`	This macro can be used in place of the `inline` keyword to instruct the compiler that a function should never be inlined. One should typically use this macro to mark functions that are unlikely to be called, such as error handling routines. Usage example: ```cpp\nBOOST_NOINLINE void handle_er↵\nror(const char* descr)\n{\n // ...\n}\n```

Macro	Description

`BOOST_LIKELY(X)` `BOOST_UNLIKELY(X)`	These macros communicate to the compiler that the conditional expression `X` is likely or unlikely to yield a positive result. The expression should result in a boolean value. The result of the macro is an integer or boolean value equivalent to the result of `X`.
	
	The macros are intended to be used in branching statements. The additional hint they provide can be used by the compiler to arrange the compiled code of the branches more effectively.
	
	Usage example:
	
	``cpp
	if (BOOST_UNLIKELY(ptr == NULL))
	  handle_error("ptr is NULL");
	``

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 XYYYZZ 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`. Has the same value as `_MSC_VER`.
`BOOST_MSVC_FULL_VER`	`<boost/config.hpp>`	Defined to a normalised 9 digit version of _MSC_FULL_VER (which sometimes only has 8 digits), the macro has the form VVMMPPPPP where VV is the major version number, MM is the minor version number, and PPPPP is the compiler build number.
`BOOST_GCC`	`<boost/config.hpp>`	Defined if the compiler is really GCC, as opposed to one of the many other compilers that also define `__GNUC__`. Has the value: `__GNUC__ * 10000 + __GNUC_MINOR__ * 100 + __GNUC_PATCHLEVEL__`.
`BOOST_INTEL`	`<boost/config.hpp>`	Defined if the compiler is an Intel compiler, takes the same value as the compiler version macro.
`BOOST_CLANG`	`<boost/config.hpp>`	Defined to 1 if the compiler is the Clang compiler.
`BOOST_WINDOWS`	`<boost/config.hpp>`	Defined if the Windows platform 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.

Macro	Header	Description
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.

Boost Deprecated Macros

The following have been deprecated; please use the replacements instead. They will be removed in a future version of boost.

Deprecated Macro	Replacement	When deprecated	When removed
BOOST_NO_0X_HDR_ARRAY	BOOST_NO_CXX11_HDR_AR-RAY	Boost 1.50	
BOOST_NO_0X_HDR_CHRONO	BOOST_NO_CXX11_HDR_CHRONO	Boost 1.50	
BOOST_NO_0X_HDR_CODECVT	BOOST_NO_CXX11_HDR_CO-DECVT	Boost 1.50	
BOOST_NO_0X_HDR_CONDI-TION_VARIABLE	BOOST_NO_CXX11_HDR_CON-DITION_VARIABLE	Boost 1.50	
BOOST_NO_0X_HDR_FOR-WARD_LIST	BOOST_NO_CXX11_HDR_FOR-WARD_LIST	Boost 1.50	
BOOST_NO_0X_HDR_FUTURE	BOOST_NO_CXX11_HDR_FU-TURE	Boost 1.50	
BOOST_NO_0X_HDR_INITIAL-IZER_LIST	BOOST_NO_CXX11_HDR_INI-TIALIZER_LIST	Boost 1.50	
BOOST_NO_INITIAL-IZER_LISTS	BOOST_NO_CXX11_HDR_INI-TIALIZER_LIST	Boost 1.50	
BOOST_NO_0X_HDR_MUTEX	BOOST_NO_CXX11_HDR_MU-TEX	Boost 1.50	
BOOST_NO_0X_HDR_RANDOM	BOOST_NO_CXX11_HDR_RAN-DOM	Boost 1.50	
BOOST_NO_0X_HDR_RATIO	BOOST_NO_CXX11_HDR_RA-TIO	Boost 1.50	
BOOST_NO_0X_HDR_REGEX	BOOST_NO_CXX11_HDR_REGEX	Boost 1.50	
BOOST_NO_0X_HDR_SYS-TEM_ERROR	BOOST_NO_CXX11_HDR_SYS-TEM_ERROR	Boost 1.50	
BOOST_NO_0X_HDR_THREAD	BOOST_NO_CXX11_HDR_THREAD	Boost 1.50	
BOOST_NO_0X_HDR_TUPLE	BOOST_NO_CXX11_HDR_TUPLE	Boost 1.50	
BOOST_NO_0X_HDR_TYPE_TRAITS	BOOST_NO_CXX11_HDR_TYPE_TRAITS	Boost 1.50	
BOOST_NO_0X_HDR_TYPEIN-DEX	BOOST_NO_CXX11_HDR_TYPEIN-DEX	Boost 1.50	
BOOST_NO_0X_HDR_UN-ORDERED_SET	BOOST_NO_CXX11_HDR_UN-ORDERED_SET	Boost 1.50	
BOOST_NO_0X_HDR_UN-ORDERED_MAP	BOOST_NO_CXX11_HDR_UN-ORDERED_MAP	Boost 1.50	
BOOST_NO_STD_UNORDERED	BOOST_NO_CXX11_HDR_UN-ORDERED_SET	Boost 1.50	

Deprecated Macro	Replacement	When deprecated	When removed
BOOST_NO_AUTO_DECLARA-TIONS	BOOST_NO_CXX11_AUTO_DE-CLARATIONS	Boost 1.51	
BOOST_NO_AUTO_MULTIDE-CLARATIONS	BOOST_NO_CXX11_AUTO_MUL-TIDECLARATIONS	Boost 1.51	
BOOST_NO_CHAR16_T	BOOST_NO_CXX11_CHAR16_T	Boost 1.51	
BOOST_NO_CHAR32_T	BOOST_NO_CXX11_CHAR32_T	Boost 1.51	
BOOST_NO_TEMPLATE_ALI-ASES	BOOST_NO_CXX11_TEM-PLATE_ALIASES	Boost 1.51	
BOOST_NO_CONSTEXPR	BOOST_NO_CXX11_CONSTEX-PR	Boost 1.51	
BOOST_NO_DECLTYPE	BOOST_NO_CXX11_DECLTYPE	Boost 1.51	
BOOST_NO_DECLTYPE_N3276	BOOST_NO_CXX11_DECL-TYPE_N3276	Boost 1.51	
BOOST_NO_DEFAULTED_FUNC-TIONS	BOOST_NO_CXX11_DEFAUL-TED_FUNCTIONS	Boost 1.51	
BOOST_NO_DELETED_FUNC-TIONS	BOOST_NO_CXX11_DE-LETED_FUNCTIONS	Boost 1.51	
BOOST_NO_EXPLICIT_CON-VERSION_OPERATORS	BOOST_NO_CXX11_EXPLI-CIT_CONVERSION_OPERAT-ORS	Boost 1.51	
BOOST_NO_EXTERN_TEM-PLATE	BOOST_NO_CXX11_EX-TERN_TEMPLATE	Boost 1.51	
BOOST_NO_FUNCTION_TEM-PLATE_DEFAULT_ARGS	BOOST_NO_CXX11_FUNC-TION_TEMPLATE_DE-FAULT_ARGS	Boost 1.51	
BOOST_NO_LAMBDAS	BOOST_NO_CXX11_LAMBDAS	Boost 1.51	
BOOST_NO_LOC-AL_CLASS_TEMPLATE_PARA-METERS	BOOST_NO_CXX11_LOC-AL_CLASS_TEMPLATE_PARA-METERS	Boost 1.51	
BOOST_NO_NOEXCEPT	BOOST_NO_CXX11_NOEXCEPT	Boost 1.51	
BOOST_NO_NULLPTR	BOOST_NO_CXX11_NULLPTR	Boost 1.51	
BOOST_NO_RAW_LITERALS	BOOST_NO_CXX11_RAW_LIT-ERALS	Boost 1.51	
BOOST_NO_RVALUE_REFER-ENCES	BOOST_NO_CXX11_RVALUE_REF-ERENCES	Boost 1.51	
BOOST_NO_SCOPED_ENUMS	BOOST_NO_CXX11_SCOPED_ENUMS	Boost 1.51	

Deprecated Macro	Replacement	When deprecated	When removed
BOOST_NO_STATIC_ASSERT	BOOST_NO_CXX11_STAT-IC_ASSERT	Boost 1.51	
BOOST_NO_STD_UNORDERED	BOOST_NO_CXX11_STD_UN-ORDERED	Boost 1.51	
BOOST_NO_UNICODE_LITER-ALS	BOOST_NO_CXX11_UNI-CODE_LITERALS	Boost 1.51	
BOOST_NO_UNIFIED_INI-TIALIZATION_SYNTAX	BOOST_NO_CXX11_UNI-FIED_INITIALIZATION_SYN-TAX	Boost 1.51	
BOOST_NO_VARIADIC_TEM-PLATES	BOOST_NO_CXX11_VARIAD-IC_TEMPLATES	Boost 1.51	
BOOST_NO_VARIADIC_MAC-ROS	BOOST_NO_CXX11_VARIAD-IC_MACROS	Boost 1.51	
BOOST_NO_NUMERIC_LIM-ITS_LOWEST	BOOST_NO_CXX11_NUMER-IC_LIMITS	Boost 1.51	
BOOST_HAS_STATIC_ASSERT	BOOST_NO_CXX11_STAT-IC_ASSERT (negated)	Boost 1.53	
BOOST_HAS_VARIADIC_TMPL	BOOST_NO_CXX11_VARIAD-IC_TEMPLATES (negated)	Boost 1.53	
BOOST_HAS_RVALUE_REFS	BOOST_NO_CXX11_RVALUE_REF-ERENCES (negated)	Boost 1.53	
BOOST_HAS_CHAR16_T	BOOST_NO_CXX11_CHAR16_T (negated)	Boost 1.53	
BOOST_HAS_CHAR32_T	BOOST_NO_CXX11_CHAR32_T (negated)	Boost 1.53	

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 several issues:

- Controlling shared library symbol visibility

- Fixing the ABI of the compiled library

- Selecting which compiled library to link against based upon the compilers settings

See Guidelines for Authors of Boost Libraries Containing Separate Source

Macros controlling shared library symbol visibility

Some compilers support C++ extensions that control which symbols will be exported from shared libraries such as dynamic shared objects (DSO's) on Unix-like systems or dynamic-link libraries (DLL's) on Windows.

The Microsoft VC++ compiler has long supplied `__declspec(dllexport)` and `__declspec(dllimport)` extensions for this purpose, as do virtually all other compilers targeting the Windows platform.

Modern versions of the GNU GCC compiler provide the `__attribute__((visibility("default")))` extension to indicate that a symbol should be exported. All other symbols may be hidden by using the `-fvisibility-hidden` or `-fvisibility-ms-compat` compiler switches.

Boost supplies several macros to make it easier to manage symbol visibility in a way that is portable between compilers and operating systems.

Macro	Description
`BOOST_SYMBOL_EXPORT`	Defines the syntax of a C++ language extension that indicates a symbol is to be exported from a shared library. If the compiler has no such extension, the macro is defined with no replacement text.
`BOOST_SYMBOL_IMPORT`	Defines the syntax of a C++ language extension that indicates a symbol is to be imported from a shared library. If the compiler has no such extension, the macro is defined with no replacement text.
| `BOOST_SYMBOL_VISIBLE` | Defines the syntax of a C++ language extension that indicates a symbol is to be globally visible. If the compiler has no such extension, the macro is defined with no replacement text. Needed for classes that are not otherwise exported, but are used by RTTI. Examples include class for objects that will be thrown as exceptions or used in dynamic_casts, across shared library boundaries. For example, a header-only exception class might look like this:<br><br>```cpp
class BOOST_SYMBOL_VISIBLE my_exception : public std::runtime_error { ... };
```<br><br>Without BOOST_SYMBOL_VISIBLE, it would be impossible to catch my_exception thrown from a shared library compiled by GCC with the -fvisibility=hidden option. |
| `BOOST_HAS_DECLSPEC` | The compiler has C++ extensions `__declspec(dllexport)` and `__declspec(dllimport)` to control export/import of symbols from shared libraries. *Deprecated. This macro is no longer necessary since BOOST_SYMBOL_EXPORT and BOOST_SYMBOL_IMPORT are now supplied. It is provided to support legacy code.* |

Typical usage:

boost/foo/config.hpp

```
...
#if defined(BOOST_ALL_DYN_LINK) || defined(BOOST_FOO_DYN_LINK)
# if defined(BOOST_FOO_SOURCE)
#   define BOOST_FOO_DECL BOOST_SYMBOL_EXPORT
# else
#   define BOOST_FOO_DECL BOOST_SYMBOL_IMPORT
# endif
#else
# define BOOST_FOO_DECL
#endif
...
```

boost/foo/foo.hpp

```
#include <boost/foo/config.hpp>
...
class BOOST_FOO_DECL bar { ... };
...
void BOOST_FOO_DECL f();
...
```

boost/libs/foo/src/foo.cpp

```
#define BOOST_FOO_SOURCE
#include <boost/foo/foo.hpp>
...
void BOOST_FOO_DECL f()
{
  ...
}
...
```

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 warnings 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.
The xxx_pass_test and the xxx_fail_test **should both report **passed****.
If *MACRONAME* is not defined when it should be defined, xxx_pass_test will not report **passed**. If *MACRONAME* is defined when it should not be defined, xxx_fail_test will not report **passed**.

• 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.