# Boost.StaticAssert

John Maddock

Steve Cleary



## Table of Contents



This manual is also available in printer friendly PDF format.

# Overview and Tutorial

The header <boost/static_assert.hpp> supplies two macros:

```
BOOST_STATIC_ASSERT(x)
BOOST_STATIC_ASSERT_MSG(x, msg)
```

Both generate a compile time error message if the integral-constant-expression x is not true. In other words, they are the compile time equivalent of the assert macro; this is sometimes known as a "compile-time-assertion", but will be called a "static assertion" throughout these docs. Note that if the condition is true, then the macros will generate neither code nor data - and the macros can also be used at either namespace, class or function scope. When used in a template, the static assertion will be evaluated at the time the template is instantiated; this is particularly useful for validating template parameters.

If the C++0x static_assert feature is available, both macros will use it. For BOOST_STATIC_ASSERT(x), the error message will be a stringized version of x. For BOOST_STATIC_ASSERT_MSG(x, msg), the error message will be the msg string.

If the C++0x static_assert feature is not available, BOOST_STATIC_ASSERT_MSG(x, msg) will be treated as BOOST_STATIC_ASSERT(x).

The material that follows assumes the C++0x static_assert feature is not available.

One of the aims of BOOST_STATIC_ASSERT is to generate readable error messages. These immediately tell the user that a library is being used in a manner that is not supported. While error messages obviously differ from compiler to compiler, but you should see something like:

```
Illegal use of STATIC_ASSERTION_FAILURE<false>
```

Which is intended to at least catch the eye!

You can use BOOST_STATIC_ASSERT at any place where you can place a declaration, that is at class, function or namespace scope, this is illustrated by the following examples:

# Use at namespace scope.

The macro can be used at namespace scope, if there is some requirement must always be true; generally this means some platform specific requirement. Suppose we require that int be at least a 32-bit integral type, and that wchar_t be an unsigned type. We can verify this at compile time as follows:

```
#include <climits>
#include <cwchar>
#include <limits>
#include <boost/static_assert.hpp>

namespace my_conditions {

   BOOST_STATIC_ASSERT(std::numeric_limits<int>::digits >= 32);
   BOOST_STATIC_ASSERT(WCHAR_MIN >= 0);

} // namespace my_conditions
```

The use of the namespace my_conditions here requires some comment. The macro BOOST_STATIC_ASSERT works by generating an typedef declaration, and since the typedef must have a name, the macro generates one automatically by mangling a stub name with the value of __LINE__. When BOOST_STATIC_ASSERT is used at either class or function scope then each use of BOOST_STATIC_ASSERT is guaranteed to produce a name unique to that scope (provided you only use the macro once on each line). However when used in a header at namespace scope, that namespace can be continued over multiple headers, each of which may have their own static assertions, and on the "same" lines, thereby generating duplicate declarations. In theory the compiler should silently ignore duplicate typedef declarations, however many do not do so (and even if they do they are entitled to emit warnings in

such cases). To avoid potential problems, if you use `BOOST_STATIC_ASSERT` in a header and at namespace scope, then enclose them in a namespace unique to that header.

# Use at function scope

The macro is typically used at function scope inside template functions, when the template arguments need checking. Imagine that we have an iterator-based algorithm that requires random access iterators. If the algorithm is instantiated with iterators that do not meet our requirements then an error will be generated eventually, but this may be nested deep inside several templates, making it hard for the user to determine what went wrong. One option is to add a static assertion at the top level of the template, in that case if the condition is not met, then an error will be generated in a way that makes it reasonably obvious to the user that the template is being misused.

```
#include <iterator>
#include <boost/static_assert.hpp>
#include <boost/type_traits.hpp>

template <class RandomAccessIterator >
RandomAccessIterator foo(RandomAccessIterator from,
                         RandomAccessIterator to)
{
   // this template can only be used with
   // random access iterators...
   typedef typename std::iterator_traits<
         RandomAccessIterator >::iterator_category cat;
   BOOST_STATIC_ASSERT(
      (boost::is_convertible<
         cat,
         const std::random_access_iterator_tag&>::value));
   //
   // detail goes here...
   return from;
}
```

A couple of footnotes are in order here: the extra set of parenthesis around the assert, is to prevent the comma inside the `is_convertible` template being interpreted by the preprocessor as a macro argument separator; the target type for `is_convertible` is a reference type, as some compilers have problems using `is_convertible` when the conversion is via a user defined constructor (in any case there is no guarantee that the iterator tag classes are copy-constructible).

# Use at class scope

The macro is typically used inside classes that are templates. Suppose we have a template-class that requires an unsigned integral type with at least 16-bits of precision as a template argument, we can achieve this using something like this:

```
#include <limits>
#include <boost/static_assert.hpp>

template <class UnsignedInt>
class myclass
{
private:
   BOOST_STATIC_ASSERT_MSG(std::numeric_limits<UnsignedInt>::is_specialized, "myclass can only ↵
be specialized for types with numeric_limits support.");
   BOOST_STATIC_ASSERT_MSG(std::numeric_limits<UnsignedInt>::digits >= 16, "Template argument ↵
UnsignedInt must have at least 16 bits precision.")
   BOOST_STATIC_ASSERT_MSG(std::numeric_limits<UnsignedInt>::is_integer, "Template argument Un↵
signedInt must be an integer.");
   BOOST_STATIC_ASSERT_MSG(!std::numeric_limits<UnsignedInt>::is_signed, "Template argument Un↵
signedInt must not be signed.");
public:
   /* details here */
};
```

# Use in templates

Normally static assertions when used inside a class or function template, will not be instantiated until the template in which it is used is instantiated. However, there is one potential problem to watch out for: if the static assertion is not dependent upon one or more template parameters, then the compiler is permitted to evaluate the static assertion at the point it is first seen, irrespective of whether the template is ever instantiated, for example:

```
template <class T>
struct must_not_be_instantiated
{
   BOOST_STATIC_ASSERT(false);
};
```

Will produce a compiler error with some compilers (for example Intel 8.1 or gcc 3.4), regardless of whether the template is ever instantiated. A workaround in cases like this is to force the assertion to be dependent upon a template parameter:

```
template <class T>
struct must_not_be_instantiated
{
   // this will be triggered if this type is instantiated
   BOOST_STATIC_ASSERT(sizeof(T) == 0);
};
```

# How it works

`BOOST_STATIC_ASSERT` works as follows. There is class `STATIC_ASSERTION_FAILURE` which is defined as:

```
namespace boost{

template <bool> struct STATIC_ASSERTION_FAILURE;

template <> struct STATIC_ASSERTION_FAILURE<true>{};

}
```

The key feature is that the error message triggered by the undefined expression `sizeof(STATIC_ASSERTION_FAILURE<0>)`, tends to be consistent across a wide variety of compilers. The rest of the machinery of `BOOST_STATIC_ASSERT` is just a way to feed the `sizeof` expression into a `typedef`. The use of a macro here is somewhat ugly; however boost members have spent considerable effort trying to invent a static assert that avoided macros, all to no avail. The general conclusion was that the good of a static assert working at namespace, function, and class scope outweighed the ugliness of a macro.

# Test Programs

**Table 1. Test programs provided with static_assert**

| Test Program | Expected to Compile | Description |
|---|---|---|
| static_assert_test.cpp | Yes | Illustrates usage, and should always compile, really just tests compiler compatibility. |
| static_assert_example_1.cpp | Platform dependent. | Namespace scope test program, may compile depending upon the platform. |
| static_assert_example_2.cpp | Yes | Function scope test program. |
| static_assert_example_3.cpp | Yes | Class scope test program. |
| static_assert_test_fail_1.cpp | No | Illustrates failure at namespace scope. |
| static_assert_test_fail_2.cpp | No | Illustrates failure at non-template function scope. |
| static_assert_test_fail_3.cpp | No | Illustrates failure at non-template class scope. |
| static_assert_test_fail_4.cpp | No | Illustrates failure at non-template class scope. |
| static_assert_test_fail_5.cpp | No | Illustrates failure at template class scope. |
| static_assert_test_fail_6.cpp | No | Illustrates failure at template class member function scope. |
| static_assert_test_fail_7.cpp | No | Illustrates failure of class scope example. |
| static_assert_test_fail_8.cpp | No | Illustrates failure of function scope example. |
| static_assert_test_fail_9.cpp | No | Illustrates failure of function scope example (part 2). |