Boost.Core

Table of Contents

Introduction

The Boost.Core library is a collection of core utilities. The criteria for inclusion is that the utility component be:

- simple,
- used by other Boost libraries, and
- not dependent on any other Boost modules except Core itself, Config, Assert, Static Assert, or Predef.

addressof

Authors

- Brad King
- Douglas Gregor
- Peter Dimov

Header <boost/core/addressof.hpp>

The header `<boost/core/addressof.hpp>` defines the function template `boost::addressof`. `boost::addressof(x)` returns the address of `x`. Ordinarily, this address can be obtained by `&x`, but the unary `&` operator can be overloaded. `boost::addressof` avoids calling used-defined `operator&()`.

`boost::addressof` was originally contributed by Brad King based on ideas from discussion with Doug Gregor.

Synopsis

```
namespace boost
{
    template<class T> T* addressof( T& x );
}
```

Example

```
#include <boost/core/addressof.hpp>

struct useless_type { };

class nonaddressable {
    useless_type operator&() const;
};

void f() {
    nonaddressable x;
    nonaddressable* xp = boost::addressof(x);
    // nonaddressable* xpe = &x; /* error */
}
```

checked_delete

Authors

- Beman Dawes
- Dave Abrahams
- Vladimir Prus
- Rainer Deyke
- John Maddock

Overview

The header <boost/checked_delete.hpp> defines two function templates, checked_delete and checked_array_delete, and two class templates, checked_deleter and checked_array_deleter.

The C++ Standard allows, in 5.3.5/5, pointers to incomplete class types to be deleted with a delete-expression. When the class has a non-trivial destructor, or a class-specific operator delete, the behavior is undefined. Some compilers issue a warning when an incomplete type is deleted, but unfortunately, not all do, and programmers sometimes ignore or disable warnings.

A particularly troublesome case is when a smart pointer's destructor, such as boost::scoped_ptr<T>::~scoped_ptr, is instantiated with an incomplete type. This can often lead to silent, hard to track failures.

The supplied function and class templates can be used to prevent these problems, as they require a complete type, and cause a compilation error otherwise.

Synopsis

```
namespace boost
{
    template<class T> void checked_delete(T * p);
    template<class T> void checked_array_delete(T * p);
    template<class T> struct checked_deleter;
    template<class T> struct checked_array_deleter;
}
```

checked_delete

template<class T> void checked_delete(T * p);

- **Requires:** T must be a complete type. The expression delete p must be well-formed.
- **Effects:** delete p;

checked_array_delete

template<class T> void checked_array_delete(T * p);

- **Requires:** T must be a complete type. The expression delete [] p must be well-formed.
- **Effects:** delete [] p;

checked_deleter

```
template<class T> struct checked_deleter
{
    typedef void result_type;
    typedef T * argument_type;
    void operator()(T * p) const;
};
```

void checked_deleter<T>::operator()(T * p) const;

- **Requires:** `T` must be a complete type. The expression `delete p` must be well-formed.
- **Effects:** `delete p;`

checked_array_deleter

```
template<class T> struct checked_array_deleter
{
    typedef void result_type;
    typedef T * argument_type;
    void operator()(T * p) const;
};
```

void checked_array_deleter<T>::operator()(T * p) const;

- **Requires:** `T` must be a complete type. The expression `delete [] p` must be well-formed.
- **Effects:** `delete [] p;`

Acknowledgements

The function templates `checked_delete` and `checked_array_delete` were originally part of `<boost/utility.hpp>`, and the documentation acknowledged Beman Dawes, Dave Abrahams, Vladimir Prus, Rainer Deyke, John Maddock, and others as contributors.

demangle

Authors

- Peter Dimov
- Andrey Semashev

Header <boost/core/demangle.hpp>

The header `<boost/core/demangle.hpp>` defines several tools for undecorating symbol names.

Synopsis

```
namespace boost
{

namespace core
{
    std::string demangle( char const * name );

    char const * demangle_alloc( char const * name ) noexcept;
    void demangle_free( char const * demangled_name ) noexcept;

    class scoped_demangled_name
    {
    public:
        explicit scoped_demangled_name( char const * name ) noexcept;
        ~scoped_demangled_name() noexcept;
        char const * get() const noexcept;

        scoped_demangled_name( scoped_demangled_name const& ) = delete;
        scoped_demangled_name& operator= ( scoped_demangled_name const& ) = delete;
    };
}

}
```

Conventional interface

The function `boost::core::demangle` is the conventional way to obtain demangled symbol name. It takes a mangled string such as those returned by `typeid(T).name()` on certain implementations such as `g++`, and returns its demangled, human-readable, form. In case if demangling fails (e.g. if `name` cannot be interpreted as a mangled name) the function returns `name`.

Example

```
#include <boost/core/demangle.hpp>
#include <typeinfo>
#include <iostream>

template<class T> struct X
{
};

int main()
{
    char const * name = typeid( X<int> ).name();

    std::cout << name << std::endl; // prints 1XIiE
    std::cout << boost::core::demangle( name ) << std::endl; // prints X<int>
}
```

Low level interface

In some cases more low level interface may be desirable. For example:

- Assuming that symbol demangling may fail, the user wants to be able to handle such errors.
- The user needs to post-process the demangled name (e.g. remove common namespaces), and allocating a temporary string with the complete demangled name is significant overhead.

The function `boost::core::demangle_alloc` performs name demangling and returns a pointer to a string with the demangled name, if succeeded, or `nullptr` otherwise. The returned pointer must be passed to `boost::core::demangle_free` to reclaim resources. Note that on some platforms the pointer returned by `boost::core::demangle_alloc` may refer to the string denoted by `name`, so this string must be kept immutable for the whole life time of the returned pointer.

The `boost::core::scoped_demangled_name` class is a scope guard that automates the calls to `boost::core::demangle_alloc` (on its construction) and `boost::core::demangle_free` (on destruction). The string with the demangled name can be obtained with its `get` method. Note that this method may return `nullptr` if demangling failed.

Example

```
#include <boost/core/demangle.hpp>
#include <typeinfo>
#include <iostream>

template<class T> struct X
{
};

int main()
{
    char const * name = typeid( X<int> ).name();
    boost::core::scoped_demangled_name demangled( name );

    std::cout << name << std::endl; // prints 1XIiE
   std::cout << (demangled.get() ? demangled.get() : "[unknown]") << std::endl; // prints X<int>
}
```

Acknowledgments

The implementation of `core::demangle` was taken from `boost/exception/detail/type_info.hpp`, which in turn was adapted from `boost/units/detail/utility.hpp` and `boost/log/utility/type_info_wrapper.hpp`.

enable_if

Authors

- Jaakko Järvi
- Jeremiah Willcock
- Andrew Lumsdaine

Introduction

The `enable_if` family of templates is a set of tools to allow a function template or a class template specialization to include or exclude itself from a set of matching functions or specializations based on properties of its template arguments. For example, one can define function templates that are only enabled for, and thus only match, an arbitrary set of types defined by a traits class. The `enable_if` templates can also be applied to enable class template specializations. Applications of `enable_if` are discussed in length in [1] and [2].

Header <boost/core/enable_if.hpp>

```
namespace boost {
    template <class Cond, class T = void> struct enable_if;
    template <class Cond, class T = void> struct disable_if;
    template <class Cond, class T> struct lazy_enable_if;
    template <class Cond, class T> struct lazy_disable_if;

    template <bool B, class T = void> struct enable_if_c;
    template <bool B, class T = void> struct disable_if_c;
    template <bool B, class T> struct lazy_enable_if_c;
    template <bool B, class T> struct lazy_disable_if_c;
}
```

Background

Sensible operation of template function overloading in C++ relies on the *SFINAE* (substitution-failure-is-not-an-error) principle [3]: if an invalid argument or return type is formed during the instantiation of a function template, the instantiation is removed from the overload resolution set instead of causing a compilation error. The following example, taken from [1], demonstrates why this is important:

```
int negate(int i) { return -i; }

template <class F>
typename F::result_type negate(const F& f) { return -f(); }
```

Suppose the compiler encounters the call `negate(1)`. The first definition is obviously a better match, but the compiler must nevertheless consider (and instantiate the prototypes) of both definitions to find this out. Instantiating the latter definition with `F` as `int` would result in:

```
int::result_type negate(const int&);
```

where the return type is invalid. If this were an error, adding an unrelated function template (that was never called) could break otherwise valid code. Due to the SFINAE principle the above example is not, however, erroneous. The latter definition of `negate` is simply removed from the overload resolution set.

The `enable_if` templates are tools for controlled creation of the SFINAE conditions.

The enable_if templates

The names of the `enable_if` templates have three parts: an optional `lazy_` tag, either `enable_if` or `disable_if`, and an optional `_c` tag. All eight combinations of these parts are supported. The meaning of the `lazy_` tag is described in the section below. The second part of the name indicates whether a true condition argument should enable or disable the current overload. The third part of the name indicates whether the condition argument is a `bool` value (`_c` suffix), or a type containing a static `bool` constant named `value` (no suffix). The latter version interoperates with Boost.MPL.

The definitions of `enable_if_c` and `enable_if` are as follows (we use `enable_if` templates unqualified but they are in the `boost` namespace).

```
template <bool B, class T = void>
struct enable_if_c {
    typedef T type;
};

template <class T>
struct enable_if_c<false, T> {};

template <class Cond, class T = void>
struct enable_if : public enable_if_c<Cond::value, T> {};
```

An instantiation of the `enable_if_c` template with the parameter `B` as `true` contains a member type `type`, defined to be `T`. If `B` is `false`, no such member is defined. Thus `enable_if_c<B, T>::type` is either a valid or an invalid type expression, depending on the value of `B`. When valid, `enable_if_c<B, T>::type` equals `T`. The `enable_if_c` template can thus be used for controlling when functions are considered for overload resolution and when they are not. For example, the following function is defined for all arithmetic types (according to the classification of the Boost **type_traits** library):

```
template <class T>
typename enable_if_c<boost::is_arithmetic<T>::value, T>::type
foo(T t) { return t; }
```

The `disable_if_c` template is provided as well, and has the same functionality as `enable_if_c` except for the negated condition. The following function is enabled for all non-arithmetic types.

```
template <class T>
typename disable_if_c<boost::is_arithmetic<T>::value, T>::type
bar(T t) { return t; }
```

For easier syntax in some cases and interoperation with Boost.MPL we provide versions of the `enable_if` templates taking any type with a `bool` member constant named `value` as the condition argument. The MPL `bool_`, `and_`, `or_`, and `not_` templates are likely to be useful for creating such types. Also, the traits classes in the Boost.Type_traits library follow this convention. For example, the above example function `foo` can be alternatively written as:

```
template <class T>
typename enable_if<boost::is_arithmetic<T>, T>::type
foo(T t) { return t; }
```

Using enable_if

The `enable_if` templates are defined in `boost/utility/enable_if.hpp`, which is included by `boost/utility.hpp`.

With respect to function templates, `enable_if` can be used in multiple different ways:

- As the return type of an instantiatied function
- As an extra parameter of an instantiated function

- As an extra template parameter (useful only in a compiler that supports C++0x default arguments for function template parameters, see Enabling function templates in C++0x for details.

In the previous section, the return type form of `enable_if` was shown. As an example of using the form of `enable_if` that works via an extra function parameter, the `foo` function in the previous section could also be written as:

```
template <class T>
T foo(T t,
    typename enable_if<boost::is_arithmetic<T> >::type* dummy = 0);
```

Hence, an extra parameter of type `void*` is added, but it is given a default value to keep the parameter hidden from client code. Note that the second template argument was not given to `enable_if`, as the default `void` gives the desired behavior.

Which way to write the enabler is largely a matter of taste, but for certain functions, only a subset of the options is possible:

- Many operators have a fixed number of arguments, thus `enable_if` must be used either in the return type or in an extra template parameter.
- Functions that have a variadic parameter list must use either the return type form or an extra template parameter.
- Constructors do not have a return type so you must use either an extra function parameter or an extra template parameter.
- Constructors that have a variadic parameter list must an extra template parameter.
- Conversion operators can only be written with an extra template parameter.

Enabling function templates in C++0x

In a compiler which supports C++0x default arguments for function template parameters, you can enable and disable function templates by adding an additional template parameter. This approach works in all situations where you would use either the return type form of `enable_if` or the function parameter form, including operators, constructors, variadic function templates, and even overloaded conversion operations.

As an example:

```
#include <boost/type_traits/is_arithmetic.hpp>
#include <boost/type_traits/is_pointer.hpp>
#include <boost/utility/enable_if.hpp>

class test
{
public:
    // A constructor that works for any argument list of size 10
    template< class... T,
        typename boost::enable_if_c< sizeof...( T ) == 10,
            int >::type = 0>
    test( T&&... );

    // A conversion operation that can convert to any arithmetic type
    template< class T,
        typename boost::enable_if< boost::is_arithmetic< T >,
            int >::type = 0>
    operator T() const;

    // A conversion operation that can convert to any pointer type
    template< class T,
        typename boost::enable_if< boost::is_pointer< T >,
            int >::type = 0>
    operator T() const;
};

int main()
{
    // Works
    test test_( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 );

    // Fails as expected
    test fail_construction( 1, 2, 3, 4, 5 );

    // Works by calling the conversion operator enabled for arithmetic types
    int arithmetic_object = test_;

    // Works by calling the conversion operator enabled for pointer types
    int* pointer_object = test_;

    // Fails as expected
    struct {} fail_conversion = test_;
}
```

Enabling template class specializations

Class template specializations can be enabled or disabled with `enable_if`. One extra template parameter needs to be added for the enabler expressions. This parameter has the default value `void`. For example:

```
template <class T, class Enable = void>
class A { ... };

template <class T>
class A<T, typename enable_if<is_integral<T> >::type> { ... };

template <class T>
class A<T, typename enable_if<is_float<T> >::type> { ... };
```

Instantiating `A` with any integral type matches the first specialization, whereas any floating point type matches the second one. All other types match the primary template. The condition can be any compile-time boolean expression that depends on the template arguments of the class. Note that again, the second argument to `enable_if` is not needed; the default (`void`) is the correct value.

Overlapping enabler conditions

Once the compiler has examined the enabling conditions and included the function into the overload resolution set, normal C++ overload resolution rules are used to select the best matching function. In particular, there is no ordering between enabling conditions. Function templates with enabling conditions that are not mutually exclusive can lead to ambiguities. For example:

```
template <class T>
typename enable_if<boost::is_integral<T>, void>::type
foo(T t) {}

template <class T>
typename enable_if<boost::is_arithmetic<T>, void>::type
foo(T t) {}
```

All integral types are also arithmetic. Therefore, say, for the call `foo(1)`, both conditions are true and both functions are thus in the overload resolution set. They are both equally good matches and thus ambiguous. Of course, more than one enabling condition can be simultaneously true as long as other arguments disambiguate the functions.

The above discussion applies to using `enable_if` in class template partial specializations as well.

Lazy enable_if

In some cases it is necessary to avoid instantiating part of a function signature unless an enabling condition is true. For example:

```
template <class T, class U> class mult_traits;

template <class T, class U>
typename enable_if<is_multipliable<T, U>,
    typename mult_traits<T, U>::type>::type
operator*(const T& t, const U& u) { ... }
```

Assume the class template `mult_traits` is a traits class defining the resulting type of a multiplication operator. The `is_multipliable` traits class specifies for which types to enable the operator. Whenever `is_multipliable<A, B>::value` is `true` for some types `A` and `B`, then `mult_traits<A, B>::type` is defined.

Now, trying to invoke (some other overload) of `operator*` with, say, operand types `C` and `D` for which `is_multipliable<C, D>::value` is `false` and `mult_traits<C, D>::type` is not defined is an error on some compilers. The SFINAE principle is not applied because the invalid type occurs as an argument to another template. The `lazy_enable_if` and `lazy_disable_if` templates (and their `_c` versions) can be used in such situations:

```
template<class T, class U>
typename lazy_enable_if<is_multipliable<T, U>,
    mult_traits<T, U> >::type
operator*(const T& t, const U& u) { ... }
```

The second argument of `lazy_enable_if` must be a class type that defines a nested type named `type` whenever the first parameter (the condition) is true.

Note

Referring to one member type or static constant in a traits class causes all of the members (type and static constant) of that specialization to be instantiated. Therefore, if your traits classes can sometimes contain invalid types, you should use two distinct templates for describing the conditions and the type mappings. In the above example, `is_multipliable<T, U>::value` defines when `mult_traits<T, U>::type` is valid.

Compiler workarounds

Some compilers flag functions as ambiguous if the only distinguishing factor is a different condition in an enabler (even though the functions could never be ambiguous). For example, some compilers (e.g. GCC 3.2) diagnose the following two functions as ambiguous:

```
template <class T>
typename enable_if<boost::is_arithmetic<T>, T>::type
foo(T t);

template <class T>
typename disable_if<boost::is_arithmetic<T>, T>::type
foo(T t);
```

Two workarounds can be applied:

- Use an extra dummy parameter which disambiguates the functions. Use a default value for it to hide the parameter from the caller. For example:

```
template <int> struct dummy { dummy(int) {} };

template <class T>
typename enable_if<boost::is_arithmetic<T>, T>::type
foo(T t, dummy<0> = 0);

template <class T>
typename disable_if<boost::is_arithmetic<T>, T>::type
foo(T t, dummy<1> = 0);
```

- Define the functions in different namespaces and bring them into a common namespace with `using` declarations:

```
namespace A {
    template <class T>
    typename enable_if<boost::is_arithmetic<T>, T>::type
    foo(T t);
}

namespace B {
    template <class T>
    typename disable_if<boost::is_arithmetic<T>, T>::type
    foo(T t);
}

using A::foo;
using B::foo;
```

  Note that the second workaround above cannot be used for member templates. On the other hand, operators do not accept extra arguments, which makes the first workaround unusable. As the net effect, neither of the workarounds are of assistance for templated operators that need to be defined as member functions (assignment and subscript operators).

Acknowledgements

We are grateful to Howard Hinnant, Jason Shirk, Paul Mensonides, and Richard Smith whose findings have influenced the library.

References

- [1] Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. Function overloading based on arbitrary properties of types. *C++ Users Journal*, 21(6):25--32, June 2003.

- [2] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Concept-controlled polymorphism. In Frank Pfennig and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 228--244. Springer Verlag, September 2003.
- [3] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

explicit_operator_bool

Authors

- Andrey Semashev

Overview

Header `<boost/core/explicit_operator_bool.hpp>` provides `BOOST_EXPLICIT_OPERATOR_BOOL()`, `BOOST_EXPLICIT_OPERATOR_BOOL_NOEXCEPT()` and `BOOST_CONSTEXPR_EXPLICIT_OPERATOR_BOOL()` compatibility helper macros that expand to an explicit conversion operator to `bool`. For compilers not supporting explicit conversion operators introduced in C++11 the macros expand to a conversion operator that implements the safe bool idiom. In case if the compiler is not able to handle safe bool idiom well the macros expand to a regular conversion operator to `bool`.

Examples

Both macros are intended to be placed within a user's class definition. The generated conversion operators will be implemented in terms of `operator!()` that should be defined by user in this class. In case of `BOOST_CONSTEXPR_EXPLICIT_OPERATOR_BOOL()` the generated conversion operator will be declared `constexpr` which requires the corresponding `operator!()` to also be `constexpr`.

```
template< typename T >
class my_ptr
{
    T* m_p;

public:
    BOOST_EXPLICIT_OPERATOR_BOOL()

    bool operator!() const
    {
        return !m_p;
    }
};
```

Now `my_ptr` can be used in conditional expressions, similarly to a regular pointer:

```
my_ptr< int > p;
if (p)
    std::cout << "true" << std::endl;
```

History

boost 1.56

- Added new macros `BOOST_EXPLICIT_OPERATOR_BOOL_NOEXCEPT` and `BOOST_CONSTEXPR_EXPLICIT_OPERATOR_BOOL` to define `noexcept` and `constexpr` operators.
- The header moved to Boost.Core.

boost 1.55

- The macro was extracted from Boost.Log.

ignore_unused

Authors

- Adam Wulkiewicz

Header <boost/core/ignore_unused.hpp>

The header `<boost/core/ignore_unused.hpp>` defines the function template `boost::ignore_unused()`. It may be used to suppress the "unused variable" or "unused local typedefs" compiler warnings when the variable or typedef can't be removed or commented out, e.g. when some blocks of the code are conditionally activated. C++11 variadic templates are used if they're supported, otherwise they're emulated with overloads.

Usage

```
boost::ignore_unused(v1, v2, v3);
boost::ignore_unused<T1, T2, T3>();
```

Example

```
int fun( int foo, int bar )
{
    boost::ignore_unused(bar);
#ifdef ENABLE_DEBUG_OUTPUT
    if ( foo < bar )
        std::cerr << "warning! foo < bar";
#endif
    return foo + 2;
}
```

Acknowledgments

`boost::ignore_unused()` was contributed by Adam Wulkiewicz.

is_same

Authors

- Peter Dimov

Header <boost/core/is_same.hpp>

The header `<boost/core/is_same.hpp>` defines the class template `boost::core::is_same<T1,T2>`. It defines a nested integral constant `value` which is `true` when `T1` and `T2` are the same type, and `false` when they are not.

In tandem with `BOOST_TEST_TRAIT_TRUE` and `BOOST_TEST_TRAIT_FALSE`, `is_same` is useful for writing tests for traits classes that have to define specific nested types.

Synopsis

```
namespace boost
{

namespace core
{
    template<class T1, class T2> struct is_same;
}

}
```

Example

```
#include <boost/core/lightweight_test_trait.hpp>
#include <boost/core/is_same.hpp>

template<class T> struct X
{
    typedef T& type;
};

using boost::core::is_same;

int main()
{
    BOOST_TEST_TRAIT_TRUE(( is_same<X<int>::type, int&> ));
    return boost::report_errors();
}
```

lightweight_test

Authors

- Peter Dimov
- Beman Dawes

Header <boost/core/lightweight_test.hpp>

The header `<boost/core/lightweight_test.hpp>` is a lightweight test framework. It's useful for writing Boost regression tests for components that are dependencies of Boost.Test.

When using `lightweight_test.hpp`, **do not forget** to `return boost::report_errors()` from `main`.

Synopsis

```
#define BOOST_TEST(expression) /*unspecified*/
#define BOOST_ERROR(message) /*unspecified*/
#define BOOST_TEST_EQ(expr1, expr2) /*unspecified*/
#define BOOST_TEST_NE(expr1, expr2) /*unspecified*/
#define BOOST_TEST_THROWS(expr, excep) /*unspecified*/

namespace boost
{
    int report_errors();
}
```

BOOST_TEST

```
BOOST_TEST(expression)
```

If expression is false increases the error count and outputs a message containing `expression`.

BOOST_ERROR

```
BOOST_ERROR(message)
```

Increases error count and outputs a message containing `message`.

BOOST_TEST_EQ

```
BOOST_TEST_EQ(expr1, expr2)
```

If `expr1 != expr2` increases the error count and outputs a message containing both expressions.

BOOST_TEST_NE

```
BOOST_TEST_NE(expr1, expr2)
```

If `expr1 == expr2` increases the error count and outputs a message containing both expressions.

BOOST_TEST_THROWS

```
BOOST_TEST_THROWS(expr, excep)
```

If `BOOST_NO_EXCEPTIONS` is **not** defined and if `expr` does not throw an exception of type `excep`, increases the error count and outputs a message containing the expression.

If `BOOST_NO_EXCEPTIONS` is defined, this macro expands to nothing and `expr` is not evaluated.

report_errors

```
int boost::report_errors()
```

Return the error count from `main`.

Example

```
#include <boost/core/lightweight_test.hpp>

int sqr( int x )
{
    return x * x;
}

int main()
{
    BOOST_TEST( sqr(2) == 4 );
    BOOST_TEST_EQ( sqr(-3), 9 );

    return boost::report_errors();
}
```

Header <boost/core/lightweight_test_trait.hpp>

The header `<boost/core/lightweight_test_trait.hpp>` defines a couple of extra macros for testing compile-time traits that return a boolean value.

Synopsis

```
#define BOOST_TEST_TRAIT_TRUE((Trait)) /*unspecified*/
#define BOOST_TEST_TRAIT_FALSE((Trait)) /*unspecified*/
```

BOOST_TEST_TRAIT_TRUE

```
BOOST_TEST_TRAIT_TRUE((Trait))
```

If `Trait::value != true` increases the error count and outputs a message containing `Trait`. Note the double set of parentheses; these enable `Trait` to contain a comma, which is common for templates.

BOOST_TEST_TRAIT_FALSE

```
BOOST_TEST_TRAIT_FALSE((Trait))
```

If `Trait::value != false` increases the error count and outputs a message containing `Trait`. Note the double set of parentheses.

Example

```
#include <boost/core/lightweight_test_trait.hpp>
#include <boost/core/is_same.hpp>

template<class T, class U> struct X
{
    typedef T type;
};

using boost::core::is_same;

int main()
{
    BOOST_TEST_TRAIT_TRUE(( is_same<X<int, long>::type, int> ));

    return boost::report_errors();
}
```

no_exceptions_support

Authors

- Pavel Vozenilek

Header <boost/core/no_exceptions_support.hpp>

The header `<boost/core/no_exceptions_support.hpp>` defines macros for use in code that needs to be portable to environments that do not have support for C++ exceptions.

Synopsis

```
#define BOOST_TRY /*unspecified*/
#define BOOST_CATCH(x) /*unspecified*/
#define BOOST_CATCH_END /*unspecified*/
#define BOOST_RETHROW /*unspecified*/
```

Example Use

```
void foo() {
  BOOST_TRY {
    ...
  } BOOST_CATCH(const std::bad_alloc&) {
      ...
      BOOST_RETHROW
  } BOOST_CATCH(const std::exception& e) {
      ...
  }
  BOOST_CATCH_END
}
```

With exception support enabled it will expand into:

```
void foo() {
  { try {
    ...
  } catch (const std::bad_alloc&) {
      ...
      throw;
  } catch (const std::exception& e) {
      ...
  }
  }
}
```

With exception support disabled it will expand into:

```
void foo() {
  { if(true) {
    ...
  } else if (false) {
      ...
  } else if (false)  {
      ...
  }
  }
}
```

noncopyable

Authors

- Dave Abrahams

Header <boost/core/noncopyable.hpp>

The header `<boost/noncopyable.hpp>` defines the class `boost::noncopyable`. It is intended to be used as a private base. `boost::noncopyable` has private (under C++03) or deleted (under C++11) copy constructor and a copy assignment operator and can't be copied or assigned; a class that derives from it inherits these properties.

`boost::noncopyable` was originally contributed by Dave Abrahams.

Synopsis

```
namespace boost
{
    class noncopyable;
}
```

Example

```
#include <boost/core/noncopyable.hpp>

class X: private boost::noncopyable
{
};
```

Rationale

Class noncopyable has protected constructor and destructor members to emphasize that it is to be used only as a base class. Dave Abrahams notes concern about the effect on compiler optimization of adding (even trivial inline) destructor declarations. He says:

“Probably this concern is misplaced, because `noncopyable` will be used mostly for classes which own resources and thus have non-trivial destruction semantics.”

With C++2011, using an optimized and trivial constructor and similar destructor can be enforced by declaring both and marking them `default`. This is done in the current implementation.

null_deleter

Authors

- Andrey Semashev

Header <boost/core/null_deleter.hpp>

The header `<boost/core/null_deleter.hpp>` defines the `boost::null_deleter` function object, which can be used as a deleter with smart pointers such as `unique_ptr` or `shared_ptr`. The deleter doesn't do anything with the pointer provided upon deallocation, which makes it useful when the pointed object is deallocated elsewhere.

Example

```
std::shared_ptr< std::ostream > make_stream()
{
    return std::shared_ptr< std::ostream >(&std::cout, boost::null_deleter());
}
```

ref

Authors

- Jaakko Järvi
- Peter Dimov
- Douglas Gregor
- Dave Abrahams
- Frank Mori Hess
- Ronald Garcia

Introduction

The Ref library is a small library that is useful for passing references to function templates (algorithms) that would usually take copies of their arguments. It defines the class template `boost::reference_wrapper<T>`, two functions `boost::ref` and `boost::cref` that return instances of `boost::reference_wrapper<T>`, a function `boost::unwrap_ref` that unwraps a `boost::reference_wrapper<T>` or returns a reference to any other type of object, and the two traits classes `boost::is_reference_wrapper<T>` and `boost::unwrap_reference<T>`.

The purpose of `boost::reference_wrapper<T>` is to contain a reference to an object of type `T`. It is primarily used to "feed" references to function templates (algorithms) that take their parameter by value.

To support this usage, `boost::reference_wrapper<T>` provides an implicit conversion to `T&`. This usually allows the function templates to work on references unmodified.

`boost::reference_wrapper<T>` is both CopyConstructible and Assignable (ordinary references are not Assignable).

The `expression boost::ref(x)` returns a `boost::reference_wrapper<X>(x)` where `X` is the type of `x`. Similarly, `boost::cref(x)` returns a `boost::reference_wrapper<X const>(x)`.

The expression `boost::unwrap_ref(x)` returns a `boost::unwrap_reference<X>::type&` where `X` is the type of `x`.

The expression `boost::is_reference_wrapper<T>::value` is `true` if `T` is a `reference_wrapper`, and `false` otherwise.

The type-expression `boost::unwrap_reference<T>::type` is `T::type` if `T` is a `reference_wrapper`, `T` otherwise.

Reference

Header <boost/core/ref.hpp>

```
namespace boost {
  template<typename T> struct is_reference_wrapper;

  template<typename T> class reference_wrapper;

  template<typename T> struct unwrap_reference;
  template<typename T> reference_wrapper< T > const ref(T &);
  template<typename T> reference_wrapper< T const  > const cref(T const &);
  template<typename T> void ref(T const &&);
  template<typename T> void cref(T const &&);
  template<typename T> unwrap_reference< T >::type & unwrap_ref(T &);
}
```

Struct template is_reference_wrapper

boost::is_reference_wrapper — Determine if a type `T` is an instantiation of `reference_wrapper`.

Synopsis

```
// In header: <boost/core/ref.hpp>

template<typename T>
struct is_reference_wrapper {

  // public data members
  static constexpr bool value;
};
```

Description

The value static constant will be true if the type `T` is a specialization of `reference_wrapper`.

Class template reference_wrapper

boost::reference_wrapper — Contains a reference to an object of type `T`.

Synopsis

```
// In header: <boost/core/ref.hpp>

template<typename T>
class reference_wrapper {
public:
  // types
  typedef T type;

  // construct/copy/destruct
  explicit reference_wrapper(T &);
  reference_wrapper(T &&) = delete;

  // public member functions
  operator T &() const;
  T & get() const;
  T * get_pointer() const;
};
```

Description

`reference_wrapper` is primarily used to "feed" references to function templates (algorithms) that take their parameter by value. It provides an implicit conversion to `T&`, which usually allows the function templates to work on references unmodified.

`reference_wrapper` public types

1. typedef T type;

   Type `T`.

`reference_wrapper` public construct/copy/destruct

1. ```
   explicit reference_wrapper(T & t);
   ```
   ```

Constructs a `reference_wrapper` object that stores a reference to `t`.

Does not throw.

2.
```
reference_wrapper(T && t) = delete;
```

Construction from a temporary object is disabled.

`reference_wrapper` public member functions

1.
```
operator T &() const;
```

Does not throw.
Returns: The stored reference.

2.
```
T & get() const;
```

Does not throw.
Returns: The stored reference.

3.
```
T * get_pointer() const;
```

Does not throw.
Returns: A pointer to the object referenced by the stored reference.

Struct template unwrap_reference

boost::unwrap_reference — Find the type in a `reference_wrapper`.

Synopsis

```
// In header: <boost/core/ref.hpp>

template<typename T>
struct unwrap_reference {
  // types
  typedef T type;
};
```

Description

The `typedef` type is `T::type` if `T` is a `reference_wrapper`, `T` otherwise.

Function template ref

boost::ref

Synopsis

```
// In header: <boost/core/ref.hpp>


template<typename T> reference_wrapper< T > const ref(T & t);
```

Description

Does not throw.

Returns: `reference_wrapper<T>(t)`

Function template cref

boost::cref

Synopsis

```
// In header: <boost/core/ref.hpp>

template<typename T> reference_wrapper< T const  > const cref(T const & t);
```

Description

Does not throw.

Returns: `reference_wrapper<T const>(t)`

Function template ref

boost::ref

Synopsis

```
// In header: <boost/core/ref.hpp>

template<typename T> void ref(T const &&);
```

Description

Construction from a temporary object is disabled.

Function template cref

boost::cref

Synopsis

```
// In header: <boost/core/ref.hpp>

template<typename T> void cref(T const &&);
```

Description

Construction from a temporary object is disabled.

Function template unwrap_ref

boost::unwrap_ref

Synopsis

```
// In header: <boost/core/ref.hpp>


template<typename T> unwrap_reference< T >::type & unwrap_ref(T & t);
```

Description

Does not throw.

Returns: `unwrap_reference<T>::type&(t)`

Acknowledgments

`ref` and `cref` were originally part of the Tuple library by Jaakko Järvi. They were "promoted to `boost::` status" by Peter Dimov because they are generally useful. Douglas Gregor and Dave Abrahams contributed `is_reference_wrapper` and `unwrap_reference`. Frank Mori Hess and Ronald Garcia contributed `boost::unwrap_ref`.

scoped_enum

Authors

- Beman Dawes
- Vicente J. Botet Escriba
- Anthony Williams

Overview

The `boost/core/scoped_enum.hpp` header contains a number of macros that can be used to generate C++11 scoped enums (7.2 [dcl.enum]) if the feature is supported by the compiler, otherwise emulate it with C++03 constructs. The `BOOST_NO_CXX11_SCOPED_ENUMS` macro from Boost.Config is used to detect the feature support in the compiler.

Some of the enumerations defined in the standard library are scoped enums.

```
enum class future_errc
{
    broken_promise,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
};
```

The user can portably declare such enumeration as follows:

```
BOOST_SCOPED_ENUM_DECLARE_BEGIN(future_errc)
{
    broken_promise,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
}
BOOST_SCOPED_ENUM_DECLARE_END(future_errc)
```

These macros allows to use `future_errc` in almost all the cases as an scoped enum.

```
future_errc ev = future_errc::no_state;
```

It is possible to specify the underlying type of the enumeration:

```
BOOST_SCOPED_ENUM_UT_DECLARE_BEGIN(future_errc, unsigned int)
{
    broken_promise,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
}
BOOST_SCOPED_ENUM_DECLARE_END(future_errc)
```

The enumeration supports explicit conversion from the underlying type.

The enumeration can be forward declared:

```
BOOST_SCOPED_ENUM_FORWARD_DECLARE(future_errc);
```

There are however some limitations. First, the emulated scoped enum is not a C++ enum, so `is_enum< future_errc >` will be `false_type`.

Second, the emulated scoped enum can not be used in switch nor in template arguments. For these cases the user needs to use some helpers. Instead of

```
switch (ev)
{
case future_errc::broken_promise:
    // ...
```

use

```
switch (boost::native_value(ev))
{
case future_errc::broken_promise:
    // ...
```

and instead of

```
template <>
struct is_error_code_enum< future_errc > :
    public true_type
{
};
```

use

```
template <>
struct is_error_code_enum< BOOST_SCOPED_ENUM_NATIVE(future_errc) > :
    public true_type
{
};
```

Lastly, explicit conversion to the underlying type should be performed with `boost::underlying_cast` instead of `static_cast`:

```
unsigned int val = boost::underlying_cast< unsigned int >(ev);
```

Here is usage example:

```
BOOST_SCOPED_ENUM_UT_DECLARE_BEGIN(algae, char)
{
    green,
    red,
    cyan
}
BOOST_SCOPED_ENUM_DECLARE_END(algae)
...
algae sample( algae::red );
void foo( algae color );
...
sample = algae::green;
foo( algae::cyan );
```

Deprecated syntax

In early versions of the header there were two ways to declare scoped enums, with different pros and cons to each. The other way used a different set of macros:

```
BOOST_SCOPED_ENUM_START(algae)
{
    green,
    red,
    cyan
};
BOOST_SCOPED_ENUM_END
...
BOOST_SCOPED_ENUM(algae) sample( algae::red );
void foo( BOOST_SCOPED_ENUM(algae) color );
...
sample = algae::green;
foo( algae::cyan );
```

Here `BOOST_SCOPED_ENUM_START` corresponds to `BOOST_SCOPED_ENUM_DECLARE_BEGIN`, `BOOST_SCOPED_ENUM_END` to `BOOST_SCOPED_ENUM_DECLARE_END` and `BOOST_SCOPED_ENUM` to `BOOST_SCOPED_ENUM_NATIVE`. Note also the semicolon before `BOOST_SCOPED_ENUM_END`.

In the current version these macros produce equivalent result to the ones described above and are considered deprecated.

Acquiring the underlying type of the enum

The header `boost/core/underlying_type.hpp` defines the metafunction `boost::underlying_type` which can be used to obtain the underlying type of the scoped enum. This metafunction has support for emulated scoped enums declared with macros in `boost/core/scoped_enum.hpp`. When native scoped enums are supported by the compiler, this metafunction is equivalent to `std::underlying_type`.

Unfortunately, there are configurations which implement scoped enums but not `std::underlying_type`. In this case `boost::underlying_type` has to be specialized by user. The macro `BOOST_NO_UNDERLYING_TYPE` is defined to indicate such cases.

Acknowledgments

This scoped enum emulation was developed by Beman Dawes, Vicente J. Botet Escriba and Anthony Williams.

Helpful comments and suggestions were also made by Kjell Elster, Phil Endecott, Joel Falcou, Mathias Gaunard, Felipe Magno de Almeida, Matt Calabrese, Daniel James and Andrey Semashev.

swap

Authors

- Niels Dekker
- Joseph Gauterin
- Steven Watanabe
- Eric Niebler

Header <boost/core/swap.hpp>

```
template<class T> void swap(T& left, T& right);
```

Introduction

The template function `boost::swap` allows the values of two variables to be swapped, using argument dependent lookup to select a specialized swap function if available. If no specialized swap function is available, `std::swap` is used.

Rationale

The generic `std::swap` function requires that the elements to be swapped are assignable and copy constructible. It is usually implemented using one copy construction and two assignments - this is often both unnecessarily restrictive and unnecessarily slow. In addition, where the generic swap implementation provides only the basic guarantee, specialized swap functions are often able to provide the no-throw exception guarantee (and it is considered best practice to do so where possible [1].

The alternative to using argument dependent lookup in this situation is to provide a template specialization of `std::swap` for every type that requires a specialized swap. Although this is legal C++, no Boost libraries use this method, whereas many Boost libraries provide specialized swap functions in their own namespaces.

`boost::swap` also supports swapping built-in arrays. Note that `std::swap` originally did not do so, but a request to add an overload of `std::swap` for built-in arrays has been accepted by the C++ Standards Committee[2].

Exception Safety

`boost::swap` provides the same exception guarantee as the underlying swap function used, with one exception; for an array of type `T[n]`, where `n > 1` and the underlying swap function for `T` provides the strong exception guarantee, `boost::swap` provides only the basic exception guarantee.

Requirements

Either:

- T must be assignable
- T must be copy constructible

Or:

- A function with the signature `swap(T&,T&)` is available via argument dependent lookup

[1] Scott Meyers, Effective C++ Third Edition, Item 25: "Consider support for a non-throwing swap"
[2] LWG Defect Report 809: std::swap should be overloaded for array types

Or:

- A template specialization of `std::swap` exists for T

Or:

- T is a built-in array of swappable elements

Portability

Several older compilers do not support argument dependent lookup. On these compilers `boost::swap` will call `std::swap`, ignoring any specialized swap functions that could be found as a result of argument dependent lookup.

Credits

- **Niels Dekker** - for implementing and documenting support for built-in arrays
- **Joseph Gauterin** - for the initial idea, implementation, tests, and documentation
- **Steven Watanabe** - for the idea to make `boost::swap` less specialized than `std::swap`, thereby allowing the function to have the name 'swap' without introducing ambiguity

typeinfo

Authors

- Peter Dimov

Header <boost/core/typeinfo.hpp>

The header `<boost/core/typeinfo.hpp>` defines a class `boost::core::typeinfo`, which is an alias for `std::type_info` when RTTI is enabled, and is a reasonable substitute when RTTI is not supported.

The macro `BOOST_CORE_TYPEID`, when applied to a type `T`, is the equivalent of `typeid(T)` and produces a reference to a const `typeinfo` object.

The function `boost::core::demangled_name` takes a `boost::core::typeinfo const & ti` and either returns `ti.name()`, when that string doesn't need to be demangled, or `boost::core::demangle(ti.name())`, when it does. The return type of `boost::core::demangled_name` is `char const*` in the first case and `std::string` in the second.

Synopsis

```
namespace boost
{

namespace core
{
    class typeinfo;
    /* char const* or std::string */ demangled_name( typeinfo const & ti );
}

}

#define BOOST_CORE_TYPEID(T) /*unspecified*/
```

Example

```
#include <boost/core/typeinfo.hpp>
#include <iostream>

template<class T1, class T2> struct X
{
};

int main()
{
    typedef X<void const*, void(*)(float)> T;

    boost::core::typeinfo const & ti = BOOST_CORE_TYPEID(T);

    std::cout << boost::core::demangled_name( ti ) << std::endl;
}
```