The Boost C++ Libraries

Release 1.32.0

November 18, 2004

CONTENTS

Chapter 1. Boost.Any

Kevlin Henney

1. Introduction

There are times when a generic (in the sense of *general* as opposed to *template−based programming*) type is needed: variables that are truly variable, accommodating values of many other more specific types rather than C++'s normal strict and static types. We can distinguish three basic kinds of generic type:

- Converting types that can hold one of a number of possible value types, e.g. `int` and `string`, and freely convert between them, for instance interpreting `5` as `"5"` or vice−versa. Such types are common in scripting and other interpreted languages. `boost::lexical_cast` supports such conversion functionality.
- Discriminated types that contain values of different types but do not attempt conversion between them, i.e. `5` is held strictly as an `int` and is not implicitly convertible either to `"5"` or to `5.0`. Their indifference to interpretation but awareness of type effectively makes them safe, generic containers of single values, with no scope for surprises from ambiguous conversions.
- Indiscriminate types that can refer to anything but are oblivious to the actual underlying type, entrusting all forms of access and interpretation to the programmer. This niche is dominated by `void *`, which offers plenty of scope for surprising, undefined behavior.

The `boost::any` class (based on the class of the same name described in "Valued Conversions" by Kevlin Henney, *C++ Report* 12(7), July/August 2000) is a variant value type based on the second category. It supports copying of any value type and safe checked extraction of that value strictly against its type. A similar design, offering more appropriate operators, can be used for a generalized function adaptor, `any_function`, a generalized iterator adaptor, `any_iterator`, and other object types that need uniform runtime treatment but support only compile−time template parameter conformance.

2. Examples

The following code demonstrates the syntax for using implicit conversions to and copying of any objects:

```
#include <list>
#include <boost/any.hpp>

using boost::any_cast;
typedef std::list<boost::any> many;

void append_int(many & values, int value)
{
    boost::any to_append = value;
    values.push_back(to_append);
}

void append_string(many & values, const std::string & value)
{
    values.push_back(value);
}

void append_char_ptr(many & values, const char * value)
{
    values.push_back(value);
}

void append_any(many & values, const boost::any & value)
{
    values.push_back(value);
}
```

```
void append_nothing(many & values)
{
    values.push_back(boost::any());
}
```

The following predicates follow on from the previous definitions and demonstrate the use of queries on any objects:

```
bool is_empty(const boost::any & operand)
{
    return operand.empty();
}

bool is_int(const boost::any & operand)
{
    return operand.type() == typeid(int);
}

bool is_char_ptr(const boost::any & operand)
{
    try
    {
        any_cast<const char *>(operand);
        return true;
    }
    catch(const boost::bad_any_cast &)
    {
        return false;
    }
}

bool is_string(const boost::any & operand)
{
    return any_cast<std::string>(&operand);
}

void count_all(many & values, std::ostream & out)
{
    out << "#empty == "
        << std::count_if(values.begin(), values.end(), is_empty) << std::endl;
    out << "#int == "
        << std::count_if(values.begin(), values.end(), is_int) << std::endl;
    out << "#const char * == "
        << std::count_if(values.begin(), values.end(), is_char_ptr) << std::endl;
    out << "#string == "
        << std::count_if(values.begin(), values.end(), is_string) << std::endl;
}
```

The following type, patterned after the OMG's Property Service, defines name−value pairs for arbitrary value types:

```
struct property
{
    property();
    property(const std::string &, const boost::any &);

    std::string name;
    boost::any value;
};

typedef std::list<property> properties;
```

The following base class demonstrates one approach to runtime polymorphism based callbacks that also require arbitrary argument types. The absence of virtual member templates requires that different solutions have different trade−offs in terms of efficiency, safety, and generality. Using a checked variant type offers one approach:

```
class consumer
{
```

```
public:
    virtual void notify(const any &) = 0;
    ...
};
```

3. Reference

3.1. *ValueType* requirements

Values are strongly informational objects for which identity is not significant, i.e. the focus is principally on their state content and any behavior organized around that. Another distinguishing feature of values is their granularity: normally fine−grained objects representing simple concepts in the system such as quantities.

As the emphasis of a value lies in its state not its identity, values can be copied and typically assigned one to another, requiring the explicit or implicit definition of a public copy constructor and public assignment operator. Values typically live within other scopes, i.e. within objects or blocks, rather than on the heap. Values are therefore normally passed around and manipulated directly as variables or through references, but not as pointers that emphasize identity and indirection.

The specific requirements on value types to be used in an `any` are:

- A *ValueType* is *CopyConstructible* [20.1.3].
- A *ValueType* is optionally *Assignable* [23.1]. The strong exception−safety guarantee is required for all forms of assignment.
- The destructor for a *ValueType* upholds the no−throw exception−safety guarantee.

3.2. Header <boost/any.hpp>

```
namespace boost {
  class bad_any_cast;
  class any;
  template<typename ValueType> ValueType any_cast(const any &);
  template<typename ValueType> const ValueType * any_cast(const any *);
  template<typename ValueType> ValueType * any_cast(any *);
}
```

Class bad_any_cast

boost::bad_any_cast — The exception thrown in the event of a failed `any_cast` of an `any` value.

Synopsis

```
class bad_any_cast : public std::bad_cast {
public:
  virtual const char * what() const;
};
```

Description

```
virtual const char * what() const;
```

Class any

boost::any — A class whose instances can hold instances of any type that satisfies ValueType requirements.

Synopsis

```
class any {
public:
  // construct/copy/destruct
  any();
  any(const any &);
  template<typename ValueType> any(const ValueType &);
  any & operator=(const any &);
  template<typename ValueType> any & operator=(const ValueType &);
  ~any();

  // modifiers
  any & swap(any &);

  // queries
  bool empty() const;
  const std::type_info & type() const;
};
```

Description

`any` construct/copy/destruct

1. `any();`

   Postconditions

   `this->empty()`
2. `any(const any & other);`

   Effects

   Copy constructor that copies content of `other` into new instance, so that any content is equivalent in both type and value to the content of `other`, or empty if `other` is empty.

   Throws

   May fail with a `std::bad_alloc` exception or any exceptions arising from the copy constructor of the contained type.
3. `template<typename ValueType> any(const ValueType & value);`

   Effects

   Makes a copy of `value`, so that the initial content of the new instance is equivalent in both type and value to `value`.

   Throws

   `std::bad_alloc` or any exceptions arising from the copy constructor of the contained type.
4. `any & operator=(const any & rhs);`

   Effects

   Copies content of `rhs` into current instance, discarding previous content, so that the new content is equivalent in both type and value to the content of `rhs`, or empty if `rhs.empty()`.

   Throws

   `std::bad_alloc` or any exceptions arising from the copy constructor of the contained type. Assignment satisfies the strong guarantee of exception safety.
5. `template<typename ValueType> any & operator=(const ValueType & rhs);`

   Effects

   Makes a copy of `rhs`, discarding previous content, so that the new content of is equivalent in both type and value to `rhs`.

   Throws

   `std::bad_alloc` or any exceptions arising from the copy constructor of the contained type. Assignment satisfies the strong guarantee of exception safety.

6. `~any();`

   Effects
   : Releases any and all resources used in management of instance.

   Throws
   : Nothing.

`any` modifiers

1. `any & swap(any & rhs);`

   Effects
   : Exchange of the contents of `*this` and `rhs`.

   Returns
   : `*this`

   Throws
   : Nothing.

`any` queries

1. `bool empty() const;`

   Returns
   : `true` if instance is empty, otherwise `false`.

   Throws
   : Will not throw.

2. `const std::type_info & type() const;`

   Returns
   : the `typeid` of the contained value if instance is non–empty, otherwise `typeid(void)`.

   Notes
   : Useful for querying against types known either at compile time or only at runtime.

Function any_cast

boost::any_cast

Custom keyword cast for extracting a value of a given type from an `any`.

Synopsis

```
template<typename ValueType> ValueType any_cast(const any & operand);
template<typename ValueType> const ValueType * any_cast(const any * operand);
template<typename ValueType> ValueType * any_cast(any * operand);
```

Description

Returns
: If passed a pointer, it returns a similarly qualified pointer to the value content if successful, otherwise null is returned. If passed a value or reference, it returns a copy of the value content if successful.

Throws
: Overloads taking an `any` pointer do not throw; the overload taking an `any` value or reference throws `bad_any_cast` if unsuccessful.

Rationale

The value/reference version returns a copy because the C++ keyword casts return copies.

4. Acknowledgements

Doug Gregor ported the documentation to the BoostBook format.

Chapter 2. Boost.Array

Nicolai Josuttis

1. Introduction

The C++ Standard Template Library STL as part of the C++ Standard Library provides a framework for processing algorithms on different kind of containers. However, ordinary arrays don't provide the interface of STL containers (although, they provide the iterator interface of STL containers).

As replacement for ordinary arrays, the STL provides class `std::vector`. However, `std::vector<>` provides the semantics of dynamic arrays. Thus, it manages data to be able to change the number of elements. This results in some overhead in case only arrays with static size are needed.

In his book, *Generic Programming and the STL*, Matthew H. Austern introduces a useful wrapper class for ordinary arrays with static size, called `block`. It is safer and has no worse performance than ordinary arrays. In *The C++ Programming Language*, 3rd edition, Bjarne Stroustrup introduces a similar class, called `c_array`, which I (Nicolai Josuttis) present slightly modified in my book *The C++ Standard Library – A Tutorial and Reference*, called `carray`. This is the essence of these approaches spiced with many feedback from boost.

After considering different names, we decided to name this class simply `array`.

Note that this class is suggested to be part of the next Technical Report, which will extend the C++ Standard (see http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1548.htm).

Class `array` fulfills most but not all of the requirements of "reversible containers" (see Section 23.1, [lib.container.requirements] of the C++ Standard). The reasons array is not an reversible STL container is because:

- No constructors are provided.
- Elements may have an undetermined initial value (see Section 3, Design Rationale).
- swap() has no constant complexity.
- size() is always constant, based on the second template argument of the type.
- The container provides no allocator support.

It doesn't fulfill the requirements of a "sequence" (see Section 23.1.1, [lib.sequence.reqmts] of the C++ Standard), except that:

- front() and back() are provided.
- operator[] and at() are provided.

2. Reference

2.1. Header <boost/array.hpp>

```
namespace boost {
  template<typename T, std::size_t N> class array;
  template<typename T, std::size_t N> void swap(array<T, N>&, array<T, N>&);
  template<typename T, std::size_t N> 
    bool operator==(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator!=(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator<(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator>(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator<=(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator>=(const array<T, N>&, const array<T, N>&);
}
```

Class template array

boost::array

STL compliant container wrapper for arrays of constant size

Synopsis

```
template<typename T, std::size_t N>
class array {
public:
  // types
  typedef T                                     value_type;
  typedef T*                                    iterator;
  typedef const T*                              const_iterator;
  typedef std::reverse_iterator<iterator>       reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
  typedef T&                                    reference;
  typedef const T&                              const_reference;
  typedef std::size_t                           size_type;
  typedef std::ptrdiff_t                        difference_type;

  // static constants
  static const size_type static_size = N;

  // construct/copy/destruct
  template<typename U> array& operator=(const array<U, N>&);

  // iterator support
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;

  // reverse iterator support
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;

  // capacity
  size_type size();
  bool empty();
  size_type max_size();

  // element access
  reference operator[](size_type);
  const_reference operator[](size_type) const;
  reference at(size_type);
  const_reference at(size_type) const;
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  const T* data() const;
  T* c_array();

  // modifiers
  void swap(array<T, N>&);
  void assign(const T&);

  T elems[N];
};

// specialized algorithms
template<typename T, std::size_t N> void swap(array<T, N>&, array<T, N>&);

// comparisons
template<typename T, std::size_t N>
  bool operator==(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator!=(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator<(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator>(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator<=(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator>=(const array<T, N>&, const array<T, N>&);
```

Description

`array` construct/copy/destruct

1. `template<typename U> array& operator=(const array<U, N>& other);`

   Effects

   `std::copy(rhs.begin(),rhs.end(), begin())`

`array` iterator support

1.
   ```
   iterator begin();
   const_iterator begin() const;
   ```

   Returns

   iterator for the first element

   Throws

   will not throw

2.
   ```
   iterator end();
   const_iterator end() const;
   ```

   Returns

   iterator for position after the last element

   Throws

   will not throw

`array` reverse iterator support

1.
   ```
   reverse_iterator rbegin();
   const_reverse_iterator rbegin() const;
   ```

   Returns
      reverse iterator for the first element of reverse iteration
2.
   ```
   reverse_iterator rend();
   const_reverse_iterator rend() const;
   ```

   Returns
      reverse iterator for position after the last element in reverse iteration

`array` capacity

1. `size_type size();`

   Returns
      `N`
2. `bool empty();`

   Returns
      `N==0`
   Throws
      will not throw
3. `size_type max_size();`

   Returns
      `N`
   Throws
      will not throw

`array` element access

1.
   ```
   reference operator[](size_type i);
   const_reference operator[](size_type i) const;
   ```

   Requires
      `i < N`
   Returns
      element with index `i`
   Throws
      will not throw.
2.
   ```
   reference at(size_type i);
   const_reference at(size_type i) const;
   ```

   Returns
      element with index `i`
   Throws
      `std::range_error` if `i >= N`
3.
   ```
   reference front();
   const_reference front() const;
   ```

Requires
: `N > 0`

Returns
: the first element

Throws
: will not throw

4.
```
reference back();
const_reference back() const;
```

Requires
: `N > 0`

Returns
: the last element

Throws
: will not throw

5. `const T* data() const;`

Returns
: `elems`

Throws
: will not throw

6. `T* c_array();`

Returns
: `elems`

Throws
: will not throw

`array` modifiers

1. `void swap(array<T, N>& other);`

Effects
: `std::swap_ranges(begin(), end(), other.begin())`

Complexity
: linear in `N`

2. `void assign(const T& value);`

Effects
: `std::fill_n(begin(), N, value)`

`array` specialized algorithms

1. `template<typename T, std::size_t N> void swap(array<T, N>& x, array<T, N>& y);`

Effects
: `x.swap(y)`

Throws
: will not throw.

`array` comparisons

1.
```
template<typename T, std::size_t N>
  bool operator==(const array<T, N>& x, const array<T, N>& y);
```

Returns
```
std::equal(x.begin(), x.end(), y.begin())
```
2.
```
template<typename T, std::size_t N>
  bool operator!=(const array<T, N>& x, const array<T, N>& y);
```

Returns
```
!(x == y)
```
3.
```
template<typename T, std::size_t N>
  bool operator<(const array<T, N>& x, const array<T, N>& y);
```

Returns
```
std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end())
```
4.
```
template<typename T, std::size_t N>
  bool operator>(const array<T, N>& x, const array<T, N>& y);
```

Returns
```
y < x
```
5.
```
template<typename T, std::size_t N>
  bool operator<=(const array<T, N>& x, const array<T, N>& y);
```

Returns
```
!(y < x)
```
6.
```
template<typename T, std::size_t N>
  bool operator>=(const array<T, N>& x, const array<T, N>& y);
```

Returns
```
!(x < y)
```

3. Design Rationale

There was an important design tradeoff regarding the constructors: We could implement array as an "aggregate" (see Section 8.5.1, [dcl.init.aggr], of the C++ Standard). This would mean:

- An array can be initialized with a brace−enclosing, comma−separated list of initializers for the elements of the container, written in increasing subscript order:

  ```
  boost::array<int,4> a = { { 1, 2, 3 } };
  ```

  Note that if there are fewer elements in the initializer list, then each remaining element gets default−initialized (thus, it has a defined value).

However, this approach has its drawbacks: **passing no initializer list means that the elements have an indetermined initial value**, because the rule says that aggregates may have:

- No user−declared constructors.
- No private or protected non−static data members.
- No base classes.
- No virtual functions.

Nevertheless, The current implementation uses this approach.

Note that for standard conforming compilers it is possible to use fewer braces (according to 8.5.1 (11) of the Standard). That is, you can initialize an array as follows:

```
boost::array<int,4> a = { 1, 2, 3 };
```

I'd appreciate any constructive feedback. **Please note: I don't have time to read all boost mails. Thus, to make sure that feedback arrives to me, please send me a copy of each mail regarding this class.**

The code is provided "as is" without expressed or implied warranty.

4. For more information...

To find more details about using ordinary arrays in C++ and the framework of the STL, see e.g.

The C++ Standard Library − A Tutorial and Reference
by Nicolai M. Josuttis
Addison Wesley Longman, 1999
ISBN 0−201−37926−0

Home Page of Nicolai Josuttis

5. Acknowledgements

Doug Gregor ported the documentation to the BoostBook format.

Chapter 3. Boost.Function

Douglas Gregor

`<dgregor -at- cs.indiana.edu>`

1. Introduction

The Boost.Function library contains a family of class templates that are function object wrappers. The notion is similar to a generalized callback. It shares features with function pointers in that both define a call interface (e.g., a function taking two integer arguments and returning a floating−point value) through which some implementation can be called, and the implementation that is invoked may change throughout the course of the program.

Generally, any place in which a function pointer would be used to defer a call or make a callback, Boost.Function can be used instead to allow the user greater flexibility in the implementation of the target. Targets can be any 'compatible' function object (or function pointer), meaning that the arguments to the interface designated by Boost.Function can be converted to the arguments of the target function object.

2. History & Compatibility Notes

- **Version 1.30.0**:
    - All features deprecated in version 1.29.0 have been removed from Boost.Function.
    - `boost::function` and `boost::functionN` objects can be assigned to 0 (semantically equivalent to calling `clear()`) and compared against 0 (semantically equivalent to calling `empty()`).
    - The Boost.Function code is now generated entirely by the Preprocessor library, so it is now possible to generate `boost::function` and `boost::functionN` class templates for any number of arguments.
    - The boost::bad_function_call exception class was introduced.
- **Version 1.29.0**: Boost.Function has been partially redesigned to minimize the interface and make it cleaner. Several seldom− or never−used features of the older Boost.Function have been deprecated and will be removed in the near future. Here is a list of features that have been deprecated, the likely impact of the deprecations, and how to adjust your code:
    - The `boost::function` class template syntax has changed. The old syntax, e.g., `boost::function<int, float, double, std::string>`, has been changed to a more natural syntax `boost::function<int (float, double, std::string)>`, where all return and argument types are encoded in a single function type parameter. Any other template parameters (e.g., the `Allocator`) follow this single parameter.

      The resolution to this change depends on the abilities of your compiler: if your compiler supports template partial specialization and can parse function types (most do), modify your code to use the newer syntax (preferable) or directly use one of the `functionN` classes whose syntax has not changed. If your compiler does not support template partial specialization or function types, you must take the latter option and use the numbered Boost.Function classes. This option merely requires changing types such as `boost::function<void, int, int>` to `boost::function2<void, int, int>` (adding the number of function arguments to the end of the class name).

      Support for the old syntax with the `boost::function` class template will persist for a short while, but will eventually be removed so that we can provide better error messages and link compatibility.

- The invocation policy template parameter (`Policy`) has been deprecated and will be removed. There is no direct equivalent to this rarely used feature.
- The mixin template parameter (`Mixin`) has been deprecated and will be removed. There is not direct equivalent to this rarely used feature.
- The `set` methods have been deprecated and will be removed. Use the assignment operator instead.

3. Tutorial

Boost.Function has two syntactical forms: the preferred form and the portable form. The preferred form fits more closely with the C++ language and reduces the number of separate template parameters that need to be considered, often improving readability; however, the preferred form is not supported on all platforms due to compiler bugs. The compatible form will work on all compilers supported by Boost.Function. Consult the table below to determine which syntactic form to use for your compiler.

Preferred syntax	**Portable syntax**
• GNU C++ 2.95.x, 3.0.x, 3.1.x • Comeau C++ 4.2.45.2 • SGI MIPSpro 7.3.0 • Intel C++ 5.0, 6.0 • Compaq's cxx 6.2 • Microsoft Visual C++ 7.1	• *Any compiler supporting the preferred syntax* • Microsoft Visual C++ 6.0, 7.0 • Borland C++ 5.5.1 • Sun WorkShop 6 update 2 C++ 5.3 • Metrowerks CodeWarrior 8.1

If your compiler does not appear in this list, please try the preferred syntax and report your results to the Boost list so that we can keep this table up–to–date.

3.1. Basic Usage

A function wrapper is defined simply by instantiating the `function` class template with the desired return type and argument types, formulated as a C++ function type. Any number of arguments may be supplied, up to some implementation–defined limit (10 is the default maximum). The following declares a function object wrapper `f` that takes two `int` parameters and returns a `float`:

Preferred syntax	**Portable syntax**
`boost::function<float (int x, int y)> f;`	`boost::function2<float, int, int> f;`

By default, function object wrappers are empty, so we can create a function object to assign to `f`:

```
struct int_div {
  float operator()(int x, int y) const { return ((float)x)/y; };
};

f = int_div();
```

Now we can use `f` to execute the underlying function object `int_div`:

```
std::cout << f(5, 3) << std::endl;
```

We are free to assign any compatible function object to `f`. If `int_div` had been declared to take two `long` operands, the implicit conversions would have been applied to the arguments without any user interference. The only limit on the types of arguments is that they be CopyConstructible, so we can even use references and arrays:

Preferred syntax

`boost::function<void (int values[], int n, int& sum, float& avg)> sum_avg;`

Portable syntax
`boost::function4<void, int*, int, int&, float&> sum_avg;`

```
void do_sum_avg(int values[], int n, int& sum, float& avg)
{
  sum = 0;
  for (int i = 0; i < n; i++)
    sum += values[i];
  avg = (float)sum / n;
}

sum_avg = &do_sum_avg;
```

Invoking a function object wrapper that does not actually contain a function object is a precondition violation, much like trying to call through a null function pointer, and will throw a bad_function_call exception). We can check for an empty function object wrapper by using it in a boolean context (it evaluates `true` if the wrapper is not empty) or compare it against `0`. For instance:

```
if (f)
  std::cout << f(5, 3) << std::endl;
else
  std::cout << "f has no target, so it is unsafe to call" << std::endl;
```

Alternatively, `empty()` method will return whether or not the wrapper is empty.

Finally, we can clear out a function target by assigning it to `0` or by calling the `clear()` member function, e.g.,

```
f = 0;
```

3.2. Free functions

Free function pointers can be considered singleton function objects with const function call operators, and can therefore be directly used with the function object wrappers:

```
float mul_ints(int x, int y) { return ((float)x) * y; }

f = &mul_ints;
```

Note that the `&` isn't really necessary unless you happen to be using Microsoft Visual C++ version 6.

3.3. Member functions

In many systems, callbacks often call to member functions of a particular object. This is often referred to as "argument binding", and is beyond the scope of Boost.Function. The use of member functions directly, however, is supported, so the following code is valid:

```
struct X {
  int foo(int);
};
```

Preferred syntax	**Portable syntax**

`boost::function<int (X*, int)> f;`
 	
`f = &X::foo;`
	`boost::function2<int, X*, int> f;`
	
	`f = &X::foo;`

`X x;`
`f(&x, 5);`
	`X x;`
	`f(&x, 5);`

Several libraries exist that support argument binding. Three such libraries are summarized below:

- Bind. This library allows binding of arguments for any function object. It is lightweight and very portable.
- The C++ Standard library. Using `std::bind1st` and `std::mem_fun` together one can bind the object of a pointer–to–member function for use with Boost.Function:

Preferred syntax	Portable syntax

`boost::function<int (int)> f;`
`X x;`
`f = std::bind1st(`
`      std::mem_fun(&X::foo), &x);`
`f(5); // Call x.foo(5)`
	`boost::function1<int, int> f;`
	`X x;`
	`f = std::bind1st(`
	`      std::mem_fun(&X::foo), &x);`
	`f(5); // Call x.foo(5)`

- The Lambda library. This library provides a powerful composition mechanism to construct function objects that uses very natural C++ syntax. Lambda requires a compiler that is reasonably conformant to the C++ standard.

3.4. References to Function Objects

In some cases it is expensive (or semantically incorrect) to have Boost.Function clone a function object. In such cases, it is possible to request that Boost.Function keep only a reference to the actual function object. This is done using the `ref` and `cref` functions to wrap a reference to a function object:

Preferred syntax	Portable syntax

`stateful_type a_function_object;`
`boost::function<int (int)> f;`
`f = boost::ref(a_function_object);`
 	
`boost::function<int (int)> f2(f);`
	`stateful_type a_function_object;`
	`boost::function1<int, int> f;`
	`f = boost::ref(a_function_object);`
	
	`boost::function1<int, int> f2(f);`

Here, `f` will not make a copy of `a_function_object`, nor will `f2` when it is targeted to `f`'s reference to `a_function_object`. Additionally, when using references to function objects, Boost.Function will not throw exceptions during assignment or construction.

3.5. Comparing Boost.Function function objects

Function object wrappers can be compared via `==` or `!=` against any function object that can be stored within the wrapper. If the function object wrapper contains a function object of that type, it will be compared against the given function object (which must be EqualityComparable). For instance:

```
int compute_with_X(X*, int);

f = &X::foo;
assert(f == &X::foo);
assert(&compute_with_X != f);
```

When comparing against an instance of `reference_wrapper`, the address of the object in the `reference_wrapper` is compared against the address of the object stored by the function object wrapper:

```
a_stateful_object so1, so2;
f = boost::ref(so1);
assert(f == boost::ref(so1));
assert(f == so1); // Only if a_stateful_object is EqualityComparable
assert(f != boost::ref(so2));
```

4. Reference

4.1. Definitions

- A function object `f` is *compatible* if for the given set of argument types `Arg1`, `Arg2`, ..., `ArgN` and a return type `ResultType`, the appropriate following function is well–formed:

  ```
  // if ResultType is not void
    ResultType foo(Arg1 arg1, Arg2 arg2, ..., ArgN argN)
    {
      return f(arg1, arg2, ..., argN);
    }

    // if ResultType is void
    ResultType foo(Arg1 arg1, Arg2 arg2, ..., ArgN argN)
    {
      f(arg1, arg2, ..., argN);
    }
  ```

  A special provision is made for pointers to member functions. Though they are not function objects, Boost.Function will adapt them internally to function objects. This requires that a pointer to member function of the form `R (X::*mf)(Arg1, Arg2, ..., ArgN) cv-quals` be adapted to a function object with the following function call operator overloads:

  ```
  template<typename P>
    R operator()(cv-quals P& x, Arg1 arg1, Arg2 arg2, ..., ArgN argN) const
    {
      return (*x).*mf(arg1, arg2, ..., argN);
    }
  ```
- A function object `f` of type `F` is *stateless* if it is a function pointer or if `boost::is_stateless<T>` is true. The construction of or copy to a Boost.Function object from a stateless function object will not cause exceptions to be thrown and will not allocate any storage.

4.2. Header <boost/function.hpp>

```
namespace boost {
  class bad_function_call;
  class function_base;
  template<typename R, typename T1, typename T2, ..., typename TN,
           typename Allocator = std::allocator<void> >
    class functionN;
  template<typename T1, typename T2, ..., typename TN, typename Allocator>
    void swap(functionN<T1, T2, ..., TN, Allocator>&,
              functionN<T1, T2, ..., TN, Allocator>&);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
           typename Functor>
    bool operator==(const functionN<T1, T2, ..., TN, Allocator>&, Functor);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
           typename Functor>
    bool operator==(Functor, const functionN<T1, T2, ..., TN, Allocator>&);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
           typename Functor>
    bool operator==(const functionN<T1, T2, ..., TN, Allocator>&,
                    reference_wrapper<Functor>);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
           typename Functor>
    bool operator==(reference_wrapper<Functor>,
                    const functionN<T1, T2, ..., TN, Allocator>&);
  template<typename T1, typename T2, ..., typename TN, typename Allocator1,
           typename U1, typename U2, ..., typename UN, typename Allocator2>
    void operator==(const functionN<T1, T2, ..., TN, Allocator1>&,
                    const functionN<U1, U2, ..., UN, Allocator2>&);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
```

```
           typename Functor>
    bool operator!=(const functionN<T1, T2, ..., TN, Allocator>&, Functor);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
           typename Functor>
    bool operator!=(Functor, const functionN<T1, T2, ..., TN, Allocator>&);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
           typename Functor>
    bool operator!=(const functionN<T1, T2, ..., TN, Allocator>&,
                    reference_wrapper<Functor>);
  template<typename T1, typename T2, ..., typename TN, typename Allocator,
           typename Functor>
    bool operator!=(reference_wrapper<Functor>,
                    const functionN<T1, T2, ..., TN, Allocator>&);
  template<typename T1, typename T2, ..., typename TN, typename Allocator1,
           typename U1, typename U2, ..., typename UN, typename Allocator2>
    void operator!=(const functionN<T1, T2, ..., TN, Allocator1>&,
                    const functionN<U1, U2, ..., UN, Allocator2>&);
  template<typename Signature, typename Allocator = std::allocator<void> >
    class function;
  template<typename Signature, typename Allocator>
    void swap(function<Signature, Allocator>&,
              function<Signature, Allocator>&);
  template<typename Signature, typename Allocator, typename Functor>
    bool operator==(const function<Signature, Allocator>&, Functor);
  template<typename Signature, typename Allocator, typename Functor>
    bool operator==(Functor, const function<Signature, Allocator>&);
  template<typename Signature, typename Allocator, typename Functor>
    bool operator==(const function<Signature, Allocator>&,
                    reference_wrapper<Functor>);
  template<typename Signature, typename Allocator, typename Functor>
    bool operator==(reference_wrapper<Functor>,
                    const function<Signature, Allocator>&);
  template<typename Signature1, typename Allocator1, typename Signature2,
           typename Allocator2>
    void operator==(const function<Signature1, Allocator1>&,
                    const function<Signature2, Allocator2>&);
  template<typename Signature, typename Allocator, typename Functor>
    bool operator!=(const function<Signature, Allocator>&, Functor);
  template<typename Signature, typename Allocator, typename Functor>
    bool operator!=(Functor, const function<Signature, Allocator>&);
  template<typename Signature, typename Allocator, typename Functor>
    bool operator!=(const function<Signature, Allocator>&,
                    reference_wrapper<Functor>);
  template<typename Signature, typename Allocator, typename Functor>
    bool operator!=(reference_wrapper<Functor>,
                    const function<Signature, Allocator>&);
  template<typename Signature1, typename Allocator1, typename Signature2,
           typename Allocator2>
    void operator!=(const function<Signature1, Allocator1>&,
                    const function<Signature2, Allocator2>&);
}
```

Class bad_function_call

boost::bad_function_call — An exception type thrown when an instance of a `function` object is empty when invoked.

Synopsis

```
class bad_function_call : public std::runtime_error {
public:
  // construct/copy/destruct
  bad_function_call();
};
```

Description

`bad_function_call` construct/copy/destruct

1. `bad_function_call();`

   Effects

   Constructs a `bad_function_call` exception object.

Class function_base

boost::function_base — The common base class for all Boost.Function objects. Objects of type function_base may not be created directly.

Synopsis

```
class function_base {
public:

  // capacity
  bool empty() const;

  // target access
  template<typename Functor> Functor* target();
  template<typename Functor> const Functor* target() const;
  template<typename Functor> bool contains(const Functor&) const;
};
```

Description

`function_base` capacity

1. `bool empty() const;`

   Returns

   `false` if `this` has a target, and `true` otherwise.

   Throws

   Will not throw.

`function_base` target access

1.
   ```
   template<typename Functor> Functor* target();
   template<typename Functor> const Functor* target() const;
   ```

   Returns

   If `this` stores a target of type `Functor`, returns the address of the target. Otherwise, returns the NULL pointer.

   Throws

   Will not throw.
2. `template<typename Functor> bool contains(const Functor& f) const;`

   Returns

   `true` if `this->target<Functor>()` is non–NULL and `function_equal(*(this->target<Functor>()), f)`

Class template functionN

boost::functionN — A set of generalized function pointers that can be used for callbacks or wrapping function objects.

Synopsis

```
template<typename R, typename T1, typename T2, ..., typename TN,
         typename Allocator = std::allocator<void> >
class functionN : public function_base {
public:
  // types
  typedef R         result_type;
  typedef Allocator allocator_type;
  typedef T1        argument_type;         // If N == 1
  typedef T1        first_argument_type;   // If N == 2
  typedef T2        second_argument_type;  // If N == 2
  typedef T1        arg1_type;
  typedef T2        arg2_type;
     .
     .
     .
  typedef TN        argN_type;

  // static constants
  static const int arity = N;

  // Lambda library support
  template<typename Args>
  struct sig {
    // types
    typedef result_type type;
  };

  // construct/copy/destruct
  functionN();
  functionN(const functionN&);
  template<typename F> functionN(F);
  functionN& operator=(const functionN&);
  ~functionN();

  // modifiers
  void swap(const functionN&);
  void clear();

  // capacity
  bool empty() const;
  operator safe_bool() const;
  bool operator!() const;

  // target access
  template<typename Functor> Functor* target();
  template<typename Functor> const Functor* target() const;
  template<typename Functor> bool contains(const Functor&) const;

  // invocation
  result_type operator()(arg1_type, arg2_type, ..., argN_type) const;
};

// specialized algorithms
template<typename T1, typename T2, ..., typename TN, typename Allocator>
  void swap(functionN<T1, T2, ..., TN, Allocator>&,
            functionN<T1, T2, ..., TN, Allocator>&);

// comparison operators
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>&, Functor);
```

```
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(Functor, const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>&,
                  reference_wrapper<Functor>);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(reference_wrapper<Functor>,
                  const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator==(const functionN<T1, T2, ..., TN, Allocator1>&,
                  const functionN<U1, U2, ..., UN, Allocator2>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>&, Functor);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(Functor, const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>&,
                  reference_wrapper<Functor>);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(reference_wrapper<Functor>,
                  const functionN<T1, T2, ..., TN, Allocator>&);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator!=(const functionN<T1, T2, ..., TN, Allocator1>&,
                  const functionN<U1, U2, ..., UN, Allocator2>&);
```

Description

Class template functionN is actually a family of related classes function0, function1, etc., up to some implementation–defined maximum. In this context, `N` refers to the number of parameters.

`functionN` construct/copy/destruct

1. `functionN();`

   Postconditions
   : `this->empty()`

   Throws
   : Will not throw.

2. `functionN(const functionN& f);`

   Postconditions
   : Contains a copy of the `f`'s target, if it has one, or is empty if `f.empty()`.

   Throws
   : Will not throw unless copying the target of `f` throws.

3. `template<typename F> functionN(F f);`

   Requires
   : F is a function object Callable from `this`.

   Postconditions
   : `*this` targets a copy of `f` if `f` is nonempty, or `this->empty()` if `f` is empty.

   Throws
   : Will not throw when `f` is a stateless function object.

4. `functionN& operator=(const functionN& f);`

   Postconditions
   : `*this` targets a copy of `f`'s target, if it has one, or is empty if `f.empty()`.

   Throws
   : Will not throw when the target of `f` is a stateless function object or a reference to the function object.

5. `~functionN();`

   Effects
   : If `!this->empty()`, destroys the target of this.

`functionN` modifiers

1. `void swap(const functionN& f);`

   Effects
   : Interchanges the targets of `*this` and `f`.

   Throws
   : Will not throw.

2. `void clear();`

   Postconditions
   : this->empty()

   Throws
   : Will not throw.

`functionN` capacity

1. `bool empty() const;`

   Returns
   : `false` if `this` has a target, and `true` otherwise.

   Throws
   : Will not throw.

2. `operator safe_bool() const;`

   Returns
   : A `safe_bool` that evaluates `false` in a boolean context when `this->empty()`, and `true` otherwise.

   Throws
   : Will not throw.

3. `bool operator!() const;`

   Returns
   : `this->empty()`

   Throws
   : Will not throw.

`functionN` target access

1.
```
template<typename Functor> Functor* target();
template<typename Functor> const Functor* target() const;
```

   Returns

If `this` stores a target of type `Functor`, returns the address of the target. Otherwise, returns the NULL pointer.

Throws
: Will not throw.

2. `template<typename Functor> bool contains(const Functor& f) const;`

   Returns
   : `true` if `this->target<Functor>()` is non-NULL and `function_equal(*(this->target<Functor>()), f)`

`functionN` invocation

1. `result_type operator()(arg1_type a1, arg2_type a2, ... , argN_type aN) const;`

   Effects
   : `f(a1, a2, ..., aN)`, where `f` is the target of `*this`.

   Returns
   : if `R` is `void`, nothing is returned; otherwise, the return value of the call to `f` is returned.

   Throws
   : `bad_function_call` if `!this->empty()`. Otherwise, may through any exception thrown by the target function `f`.

`functionN` specialized algorithms

1.
```
template<typename T1, typename T2, ..., typename TN, typename Allocator>
  void swap(functionN<T1, T2, ..., TN, Allocator>& f1,
            functionN<T1, T2, ..., TN, Allocator>& f2);
```

   Effects
   : `f1.swap(f2)`

   Throws
   : Will not throw.

`functionN` comparison operators

1.
```
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>& f, Functor g);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(Functor g, const functionN<T1, T2, ..., TN, Allocator>& f);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(const functionN<T1, T2, ..., TN, Allocator>& f,
                  reference_wrapper<Functor> g);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator==(reference_wrapper<Functor> g,
                  const functionN<T1, T2, ..., TN, Allocator>& f);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator==(const functionN<T1, T2, ..., TN, Allocator1>& f1,
                  const functionN<U1, U2, ..., UN, Allocator2>& f2);
```

   Returns
   : True when `f` stores an object of type `Functor` and one of the following conditions applies:

- g is of type reference_wrapper<Functor> and f.target<Functor>() == g.get_pointer().
- g is not of type reference_wrapper<Functor> and function_equal(*(f.target<Functor>()), g).

Notes

functionN objects are not EqualityComparable.

Rationale

The safe_bool conversion opens a loophole whereby two functionN instances can be compared via ==, although this is not feasible to implement. The undefined void operator== closes the loophole and ensures a compile–time or link–time error.

2.

```
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>& f, Functor g);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(Functor g, const functionN<T1, T2, ..., TN, Allocator>& f);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(const functionN<T1, T2, ..., TN, Allocator>& f,
                  reference_wrapper<Functor> g);
template<typename T1, typename T2, ..., typename TN, typename Allocator,
         typename Functor>
  bool operator!=(reference_wrapper<Functor> g,
                  const functionN<T1, T2, ..., TN, Allocator>& f);
template<typename T1, typename T2, ..., typename TN, typename Allocator1,
         typename U1, typename U2, ..., typename UN, typename Allocator2>
  void operator!=(const functionN<T1, T2, ..., TN, Allocator1>& f1,
                  const functionN<U1, U2, ..., UN, Allocator2>& f2);
```

Returns

True when f does not store an object of type Functor or it stores an object of type Functor and one of the following conditions applies:

- g is of type reference_wrapper<Functor> and f.target<Functor>() != g.get_pointer().
- g is not of type reference_wrapper<Functor> and !function_equal(*(f.target<Functor>()), g).

Notes

functionN objects are not EqualityComparable.

Rationale

The safe_bool conversion opens a loophole whereby two functionN instances can be compared via !=, although this is not feasible to implement. The undefined void operator!= closes the loophole and ensures a compile–time or link–time error.

Class template function

boost::function — A generalized function pointer that can be used for callbacks or wrapping function objects.

Synopsis

```
template<typename Signature,   // Function type R (T1, T2, ..., TN)
         typename Allocator = std::allocator<void> >
class function : public functionN<R, T1, T2, ..., TN, Allocator> {
public:
  // types
  typedef R         result_type;
  typedef Allocator allocator_type;
  typedef T1        argument_type;         // If N == 1
  typedef T1        first_argument_type;   // If N == 2
  typedef T2        second_argument_type;  // If N == 2
  typedef T1        arg1_type;
  typedef T2         arg2_type;
     .
     .
     .
  typedef TN         argN_type;

  // static constants
  static const int arity = N;

  // Lambda library support
  template<typename Args>
  struct sig {
    // types
    typedef result_type type;
  };

  // construct/copy/destruct
  function();
  function(const functionN&);
  function(const function&);
  template<typename F> function(F);
  function& operator=(const functionN&);
  function& operator=(const function&);
  ~function();

  // modifiers
  void swap(const function&);
  void clear();

  // capacity
  bool empty() const;
  operator safe_bool() const;
  bool operator!() const;

  // target access
  template<typename Functor> Functor* target();
  template<typename Functor> const Functor* target() const;
  template<typename Functor> bool contains(const Functor&) const;

  // invocation
  result_type operator()(arg1_type, arg2_type, ..., argN_type) const;
};

// specialized algorithms
template<typename Signature, typename Allocator>
  void swap(function<Signature, Allocator>&, function<Signature, Allocator>&);

// comparison operators
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>&, Functor);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(Functor, const function<Signature, Allocator>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>&,
                  reference_wrapper<Functor>);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(reference_wrapper<Functor>,
                  const function<Signature, Allocator>&);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator==(const function<Signature1, Allocator1>&,
                  const function<Signature2, Allocator2>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>&, Functor);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(Functor, const function<Signature, Allocator>&);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>&,
                    reference_wrapper<Functor>);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(reference_wrapper<Functor>,
                  const function<Signature, Allocator>&);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator!=(const function<Signature1, Allocator1>&,
                  const function<Signature2, Allocator2>&);
```

Description

Class template function is a thin wrapper around the numbered class templates function0, function1, etc. It accepts a function type with N arguments and will will derive from functionN instantiated with the arguments it receives.

The semantics of all operations in class template function are equivalent to that of the underlying functionN object, although additional member functions are required to allow proper copy construction and copy assignment of function objects.

`function` construct/copy/destruct

1. `function();`

   Postconditions
      `this->empty()`
   Throws
      Will not throw.
2. `function(const functionN& f);`

   Postconditions
      Contains a copy of the `f`'s target, if it has one, or is empty if `f.empty()`.
   Throws
      Will not throw unless copying the target of `f` throws.
3. `function(const function& f);`

   Postconditions
      Contains a copy of the `f`'s target, if it has one, or is empty if `f.empty()`.
   Throws
      Will not throw unless copying the target of `f` throws.
4. `template<typename F> function(F f);`

   Requires
      F is a function object Callable from `this`.
   Postconditions
      `*this` targets a copy of `f` if `f` is nonempty, or `this->empty()` if `f` is empty.
   Throws
      Will not throw when `f` is a stateless function object.
5. `function& operator=(const functionN& f);`

   Postconditions
      `*this` targets a copy of `f`'s target, if it has one, or is empty if `f.empty()`
   Throws
      Will not throw when the target of `f` is a stateless function object or a reference to the function object.
6. `function& operator=(const function& f);`

   Postconditions
      `*this` targets a copy of `f`'s target, if it has one, or is empty if `f.empty()`
   Throws
      Will not throw when the target of `f` is a stateless function object or a reference to the function object.

7. `~function();`

   Effects

   If `!this->empty()`, destroys the target of `this`.

`function` modifiers

1. `void swap(const function& f);`

   Effects

   Interchanges the targets of `*this` and `f`.

   Throws

   Will not throw.
2. `void clear();`

   Postconditions

   `this->empty()`

   Throws

   Will not throw.

`function` capacity

1. `bool empty() const;`

   Returns

   `false` if `this` has a target, and `true` otherwise.

   Throws

   Will not throw.
2. `operator safe_bool() const;`

   Returns

   A `safe_bool` that evaluates `false` in a boolean context when `this->empty()`, and `true` otherwise.

   Throws

   Will not throw.
3. `bool operator!() const;`

   Returns

   `this->empty()`

   Throws

   Will not throw.

`function` target access

1.
   ```
   template<typename Functor> Functor* target();
   template<typename Functor> const Functor* target() const;
   ```

   Returns

   If `this` stores a target of type `Functor`, returns the address of the target. Otherwise, returns the NULL pointer.

   Throws

   Will not throw.
2. `template<typename Functor> bool contains(const Functor& f) const;`

   Returns

`true` if `this->target<Functor>()` is non–NULL and `function_equal(*(this->target<Functor>()), f)`

`function` invocation

1.
```
result_type operator()(arg1_type a1, arg2_type a2, ... , argN_type aN) const;
```

Effects
: `f(a1, a2, ..., aN)`, where `f` is the target of `*this`.

Returns
: if `R` is `void`, nothing is returned; otherwise, the return value of the call to `f` is returned.

Throws
: `bad_function_call` if `!this->empty()`. Otherwise, may through any exception thrown by the target function `f`.

`function` specialized algorithms

1.
```
template<typename Signature, typename Allocator>
  void swap(function<Signature, Allocator>& f1,
            function<Signature, Allocator>& f2);
```

Effects
: `f1.swap(f2)`

Throws
: Will not throw.

`function` comparison operators

1.
```
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>& f, Functor g);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(Functor g, const function<Signature, Allocator>& f);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(const function<Signature, Allocator>& f,
                  reference_wrapper<Functor> g);
template<typename Signature, typename Allocator, typename Functor>
  bool operator==(reference_wrapper<Functor> g,
                  const function<Signature, Allocator>& f);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator==(const function<Signature1, Allocator1>& f1,
                  const function<Signature2, Allocator2>& f2);
```

Returns
: True when `f` stores an object of type `Functor` and one of the following conditions applies:
  - `g` is of type `reference_wrapper<Functor>` and `f.target<Functor>() == g.get_pointer()`.
  - `g` is not of type `reference_wrapper<Functor>` and `function_equals(*(f.target<Functor>()), g)`.

Notes
: `function` objects are not EqualityComparable.

Rationale
: The `safe_bool` conversion opens a loophole whereby two `function` instances can be compared via `==`, although this is not feasible to implement. The undefined `void operator==` closes the loophole and ensures a compile–time or link–time error.

2.
```
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>& f, Functor g);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(Functor g, const function<Signature, Allocator>& f);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(const function<Signature, Allocator>& f,
                  reference_wrapper<Functor> g);
template<typename Signature, typename Allocator, typename Functor>
  bool operator!=(reference_wrapper<Functor> g,
                  const function<Signature, Allocator>& f);
template<typename Signature1, typename Allocator1, typename Signature2,
         typename Allocator2>
  void operator!=(const function<Signature1, Allocator1>& f1,
                  const function<Signature2, Allocator2>& f2);
```

Returns

True when `f` does not store an object of type `Functor` or it stores an object of type `Functor` and one of the following conditions applies:

- `g` is of type `reference_wrapper<Functor>` and `f.target<Functor>() != g.get_pointer()`.
- `g` is not of type `reference_wrapper<Functor>` and `!function_equals(*(f.target<Functor>()), g)`.

Notes

`function` objects are not EqualityComparable.

Rationale

The `safe_bool` conversion opens a loophole whereby two `function` instances can be compared via `!=`, although this is not feasible to implement. The undefined `void operator!=` closes the loophole and ensures a compile–time or link–time error.

4.3. Header <boost/function_equal.hpp>

```
namespace boost {
  template<typename F, typename G> bool function_equal(const F&, const G&);
}
```

Function template function_equal

boost::function_equal

Compare two function objects for equality.

Synopsis

```
template<typename F, typename G> bool function_equal(const F& f, const G& g);
```

Description

Returns

`f == g`.

Throws

Only if `f == g` throws.

5. Frequently Asked Questions

5.1. Why can't I compare boost::function objects with `operator==` or `operator!=`?

Comparison between boost::function objects cannot be implemented "well", and therefore will not be implemented. The typical semantics requested for `f == g` given boost::function objects `f` and `g` are:

- If `f` and `g` store function objects of the same type, use that type's `operator==` to compare them.
- If `f` and `g` store function objects of different types, return `false`.

The problem occurs when the type of the function objects stored by both `f` and `g` doesn't have an `operator==`: we would like the expression `f == g` to fail to compile, as occurs with, e.g., the standard containers. However, this is not implementable for boost::function because it necessarily "erases" some type information after it has been assigned a function object, so it cannot try to call `operator==` later: it must either find a way to call `operator==` now, or it will never be able to call it later. Note, for instance, what happens if you try to put a `float` value into a boost::function object: you will get an error at the assignment operator or constructor, not in `operator()`, because the function−call expression must be bound in the constructor or assignment operator.

The most promising approach is to find a method of determining if `operator==` can be called for a particular type, and then supporting it only when it is available; in other situations, an exception would be thrown. However, to date there is no known way to detect if an arbitrary operator expression `f == g` is suitably defined. The best solution known has the following undesirable qualities:

1. Fails at compile−time for objects where `operator==` is not accessible (e.g., because it is `private`).
2. Fails at compile−time if calling `operator==` is ambiguous.
3. Appears to be correct if the `operator==` declaration is correct, even though `operator==` may not compile.

All of these problems translate into failures in the boost::function constructors or assignment operator, *even if the user never invokes operator==*. We can't do that to users.

The other option is to place the burden on users that want to use `operator==`, e.g., by providing an `is_equality_comparable` trait they may specialize. This is a workable solution, but is dangerous in practice, because forgetting to specialize the trait will result in unexpected exceptions being thrown from boost::function's `operator==`. This essentially negates the usefulness of `operator==` in the context in which it is most desired: multitarget callbacks. The Signals library has a way around this.

5.2. I see void pointers; is this [mess] type safe?

Yes, `boost::function` is type safe even though it uses void pointers and pointers to functions returning void and taking no arguments. Essentially, all type information is encoded in the functions that manage and invoke function pointers and function objects. Only these functions are instantiated with the exact type that is pointed to by the void pointer or pointer to void function. The reason that both are required is that one may cast between void pointers and object pointers safely or between different types of function pointers (provided you don't invoke a function pointer with the wrong type).

5.3. Why are there workarounds for void returns? C++ allows them!

Void returns are permitted by the C++ standard, as in this code snippet:

```
void f();
void g() { return f(); }
```

This is a valid usage of `boost::function` because void returns are not used. With void returns, we would attempting to compile ill−formed code similar to:

```
int f();
void g() { return f(); }
```

In essence, not using void returns allows `boost::function` to swallow a return value. This is consistent with allowing the user to assign and invoke functions and function objects with parameters that don't exactly match.

5.4. Why (function) cloning?

In November and December of 2000, the issue of cloning vs. reference counting was debated at length and it was decided that cloning gave more predictable semantics. I won't rehash the discussion here, but if it cloning is incorrect for a particular application a reference−counting allocator could be used.

5.5. How much overhead does a call through `boost::function` incur?

The cost of `boost::function` can be reasonably consistently measured at around 20ns +/− 10 ns on a modern >2GHz platform versus directly inlining the code.

However, the performance of your application may benefit from or be disadvantaged by `boost::function` depending on how your C++ optimiser optimises. Similar to a standard function pointer, differences of order of 10% have been noted to the benefit or disadvantage of using `boost::function` to call a function that contains a tight loop depending on your compilation circumstances.

[Answer provided by Matt Hurd. See http://article.gmane.org/gmane.comp.lib.boost.devel/33278]

6. Miscellaneous Notes

6.1. Boost.Function vs. Function Pointers

Boost.Function has several advantages over function pointers, namely:

- Boost.Function allows arbitrary compatible function objects to be targets (instead of requiring an exact function signature).
- Boost.Function may be used with argument−binding and other function object construction libraries.
- Boost.Function has predictible behavior when an empty function object is called.

And, of course, function pointers have several advantages over Boost.Function:

- Function pointers are smaller (the size of one pointer instead of three)
- Function pointers are faster (Boost.Function may require two calls through function pointers)
- Function pointers are backward−compatible with C libraries.
- More readable error messages.

6.2. Performance

6.2.1. Function object wrapper size

Function object wrappers will be the size of two function pointers plus one function pointer or data pointer (whichever is larger). On common 32−bit platforms, this amounts to 12 bytes per wrapper. Additionally, the function object target will be allocated on the heap.

6.2.2. Copying efficiency

Copying function object wrappers may require allocating memory for a copy of the function object target. The default allocator may be replaced with a faster custom allocator or one may choose to allow the function object wrappers to only store function object targets by reference (using `ref`) if the cost of this cloning becomes prohibitive.

6.2.3. Invocation efficiency

With a properly inlining compiler, an invocation of a function object requires one call through a function pointer. If the call is to a free function pointer, an additional call must be made to that function pointer (unless the compiler has very powerful interprocedural analysis).

6.3. Combatting virtual function "bloat"

The use of virtual functions tends to cause 'code bloat' on many compilers. When a class contains a virtual function, it is necessary to emit an additional function that classifies the type of the object. It has been our experience that these auxiliary functions increase the size of the executable significantly when many `boost::function` objects are used.

In Boost.Function, an alternative but equivalent approach was taken using free functions instead of virtual functions. The Boost.Function object essentially holds two pointers to make a valid target call: a void pointer to the function object it contains and a void pointer to an "invoker" that can call the function object, given the function pointer. This invoker function performs the argument and return value conversions Boost.Function provides. A third pointer points to a free function called the "manager", which handles the cloning and destruction of function objects. The scheme is typesafe because the only functions that actually handle the function object, the invoker and the manager, are instantiated given the type of the function object, so they can safely cast the incoming void pointer (the function object pointer) to the appropriate type.

6.4. Acknowledgements

Many people were involved in the construction of this library. William Kempf, Jesse Jones and Karl Nelson were all extremely helpful in isolating an interface and scope for the library. John Maddock managed the formal review, and many reviewers gave excellent comments on interface, implementation, and documentation. Peter Dimov led us to the function declarator−based syntax.

7. Testsuite

7.1. Acceptance tests

Test	Type	Description	If failing...
function_test.cpp	run	Test the capabilities of the boost::function class template.	The boost::function class template may not be usable on your compiler. However, the library may still be usable via the boost::functionN class templates.
function_n_test.cpp	run	Test the capabilities of the boost::functionN class templates.	
allocator_test.cpp	run	Test the use of custom allocators.	Allocators are ignored by the implementation.
stateless_test.cpp	run	Test the optimization of stateless function objects in the Boost.Function library.	The exception−safety and performance guarantees given for stateless function objects may not be met by the implementation.
lambda_test.cpp	run	Test the interaction between Boost.Function and Boost.Lambda.	Either Boost.Lambda does not work on the platform, or Boost.Function cannot safely be applied without the use of boost::unlambda.
contains_test.cpp	run	Test the operation of the `target` member function and the equality operators.	
function_30.cpp	compile	Test the generation of a Boost.Function function object adaptor accepting 30 arguments.	The Boost.Function library may work for function object adaptors of up to 10 parameters, but will be unable to

			generate adaptors for an arbitrary number of parameters. Failure often indicates an error in the compiler's preprocessor.
function_arith_cxx98.cpp	run	Test the first tutorial example.	
function_arith_portable.cpp	run	Test the first tutorial example.	
sum_avg_cxx98.cpp	run	Test the second tutorial example.	
sum_avg_portable.cpp	run	Test the second tutorial example.	
mem_fun_cxx98.cpp	run	Test member function example from tutorial.	
mem_fun_portable.cpp	run	Test member function example from tutorial.	
std_bind_cxx98.cpp	run	Test standard binders example from tutorial.	
std_bind_portable.cpp	run	Test standard binders example from tutorial.	
function_ref_cxx98.cpp	run	Test boost::ref example from tutorial.	
function_ref_portable.cpp	run	Test boost::ref example from tutorial.	

7.2. Negative tests

Test	Type	Description	If failing...
function_test_fail1.cpp	compile−fail	Test the (incorrect!) use of comparisons between Boost.Function function objects.	Intuitive (but incorrect!) code may compile and will give meaningless results.
function_test_fail2.cpp	compile−fail	Test the use of an incompatible function object with Boost.Function	Incorrect code may compile (with potentially unexpected results).

Chapter 4. Boost.Lambda

Jaakko Järvi

<jarvi at cs tamu edu>

1. In a nutshell

The Boost Lambda Library (BLL in the sequel) is a C++ template library, which implements form of *lambda abstractions* for C++. The term originates from functional programming and lambda calculus, where a lambda abstraction defines an unnamed function. The primary motivation for the BLL is to provide flexible and convenient means to define unnamed function objects for STL algorithms. In explaining what the library is about, a line of code says more than a thousand words; the following line outputs the elements of some STL container `a` separated by spaces:

```
for_each(a.begin(), a.end(), std::cout << _1 << ' ');
```

The expression `std::cout << _1 << ' '` defines a unary function object. The variable `_1` is the parameter of this function, a *placeholder* for the actual argument. Within each iteration of `for_each`, the function is called with an element of `a` as the actual argument. This actual argument is substituted for the placeholder, and the body of the function is evaluated.

The essence of BLL is letting you define small unnamed function objects, such as the one above, directly on the call site of an STL algorithm.

2. Getting Started

2.1. Installing the library

The library consists of include files only, hence there is no installation procedure. The `boost` include directory must be on the include path. There are a number of include files that give different functionality:

- `lambda/lambda.hpp` defines lambda expressions for different C++ operators, see Section 5.2, Operator expressions .
- `lambda/bind.hpp` defines `bind` functions for up to 9 arguments, see Section 5.3, Bind expressions .
- `lambda/if.hpp` defines lambda function equivalents for if statements and the conditional operator, see Section 5.6, Lambda expressions for control structures (includes `lambda.hpp`).
- `lambda/loops.hpp` defines lambda function equivalent for looping constructs, see Section 5.6, Lambda expressions for control structures .
- `lambda/switch.hpp` defines lambda function equivalent for the switch statement, see Section 5.6, Lambda expressions for control structures .
- `lambda/construct.hpp` provides tools for writing lambda expressions with constructor, destructor, new and delete invocations, see Section 5.8, Construction and destruction (includes `lambda.hpp`).
- `lambda/casts.hpp` provides lambda versions of different casts, as well as `sizeof` and `typeid`, see Section 5.10.1, Cast expressions .
- `lambda/exceptions.hpp` gives tools for throwing and catching exceptions within lambda functions, Section 5.7, Exceptions (includes `lambda.hpp`).
- `lambda/algorithm.hpp` and `lambda/numeric.hpp` (cf. standard `algortihm` and `numeric` headers) allow nested STL algorithm invocations, see Section 5.11, Nesting STL algorithm invocations .

Any other header files in the package are for internal use. Additionally, the library depends on two other Boost Libraries, the *Tuple*[tuple] and the *type_traits*[type_traits] libraries, and on the `boost/ref.hpp` header.

All definitions are placed in the namespace `boost::lambda` and its subnamespaces.

2.2. Conventions used in this document

In most code examples, we omit the namespace prefixes for names in the `std` and `boost::lambda` namespaces. Implicit using declarations

```
using namespace std;
using namespace boost::lambda;
```

are assumed to be in effect.

3. Introduction

3.1. Motivation

The Standard Template Library (STL) [STL94], now part of the C++ Standard Library [C++98], is a generic container and algorithm library. Typically STL algorithms operate on container elements via *function objects*. These function objects are passed as arguments to the algorithms.

Any C++ construct that can be called with the function call syntax is a function object. The STL contains predefined function objects for some common cases (such as `plus`, `less` and `not1`). As an example, one possible implementation for the standard `plus` template is:

```
template <class T> : public binary_function<T, T, T>
struct plus {
  T operator()(const T& i, const T& j) const {
    return i + j;
  }
};
```

The base class `binary_function<T, T, T>` contains typedefs for the argument and return types of the function object, which are needed to make the function object *adaptable*.

In addition to the basic function object classes, such as the one above, the STL contains *binder* templates for creating a unary function object from an adaptable binary function object by fixing one of the arguments to a constant value. For example, instead of having to explicitly write a function object class like:

```
class plus_1 {
  int _i;
public:
  plus_1(const int& i) : _i(i) {}
  int operator()(const int& j) { return _i + j; }
};
```

the equivalent functionality can be achieved with the `plus` template and one of the binder templates (`bind1st`). E.g., the following two expressions create function objects with identical functionalities; when invoked, both return the result of adding `1` to the argument of the function object:

```
plus_1(1)
bind1st(plus<int>(), 1)
```

The subexpression `plus<int>()` in the latter line is a binary function object which computes the sum of two integers, and `bind1st` invokes this function object partially binding the first argument to `1`. As an example of using the above function object, the following code adds `1` to each element of some container `a` and outputs the results into the standard output stream `cout`.

```
transform(a.begin(), a.end(), ostream_iterator<int>(cout),
          bind1st(plus<int>(), 1));
```

To make the binder templates more generally applicable, the STL contains *adaptors* for making pointers or references to functions, and pointers to member functions, adaptable. Finally, some STL implementations contain function composition operations as extensions to the standard [SGI02].

All these tools aim at one goal: to make it possible to specify *unnamed functions* in a call of an STL algorithm, in other words, to pass code fragments as an argument to a function. However, this goal is attained only partially. The simple example above shows that the definition of unnamed functions with the standard tools is cumbersome. Complex expressions involving functors, adaptors, binders and function composition operations tend to be difficult to comprehend. In addition to this, there are significant restrictions in applying the standard tools. E.g. the standard binders allow only one argument of a binary function to be bound; there are no binders for 3−ary, 4−ary etc. functions.

The Boost Lambda Library provides solutions for the problems described above:

- Unnamed functions can be created easily with an intuitive syntax. The above example can be written as:

  ```
  transform(a.begin(), a.end(), ostream_iterator<int>(cout),
            1 + _1);
  ```

  or even more intuitively:

  ```
  for_each(a.begin(), a.end(), cout << (1 + _1));
  ```
- Most of the restrictions in argument binding are removed, arbitrary arguments of practically any C++ function can be bound.
- Separate function composition operations are not needed, as function composition is supported implicitly.

3.2. Introduction to lambda expressions

Lambda expression are common in functional programming languages. Their syntax varies between languages (and between different forms of lambda calculus), but the basic form of a lambda expressions is:

$\texttt{lambda } x_1 \texttt{ ... } x_n\texttt{.e}$

A lambda expression defines an unnamed function and consists of:

- the parameters of this function: $x_1 \ldots x_n$.
- the expression e which computes the value of the function in terms of the parameters $x_1 \ldots x_n$.

A simple example of a lambda expression is

```
lambda x y.x+y
```

Applying the lambda function means substituting the formal parameters with the actual arguments:

```
(lambda x y.x+y) 2 3 = 2 + 3 = 5
```

In the C++ version of lambda expressions the $\texttt{lambda } x_1 \ldots x_n$ part is missing and the formal parameters have predefined names. In the current version of the library, there are three such predefined formal parameters, called *placeholders*: `_1`, `_2` and `_3`. They refer to the first, second and third argument of the function defined by the lambda expression. For example, the C++ version of the definition

```
lambda x y.x+y
```

is

```
_1 + _2
```

Hence, there is no syntactic keyword for C++ lambda expressions. The use of a placeholder as an operand implies that the operator invocation is a lambda expression. However, this is true only for operator invocations. Lambda expressions containing function calls, control structures, casts etc. require special syntactic constructs. Most importantly, function calls need to be wrapped inside a `bind` function. As an example, consider the lambda expression:

```
lambda x y.foo(x,y)
```

Rather than `foo(_1, _2)`, the C++ counterpart for this expression is:

```
bind(foo, _1, _2)
```

We refer to this type of C++ lambda expressions as *bind expressions*.

A lambda expression defines a C++ function object, hence function application syntax is like calling any other function object, for instance: `(_1 + _2)(i, j)`.

3.2.1. Partial function application

A bind expression is in effect a *partial function application*. In partial function application, some of the arguments of a function are bound to fixed values. The result is another function, with possibly fewer arguments. When called with the unbound arguments, this new function invokes the original function with the merged argument list of bound and unbound arguments.

3.2.2. Terminology

A lambda expression defines a function. A C++ lambda expression concretely constructs a function object, *a functor*, when evaluated. We use the name *lambda functor* to refer to such a function object. Hence, in the terminology adopted here, the result of evaluating a lambda expression is a lambda functor.

4. Using the library

The purpose of this section is to introduce the basic functionality of the library. There are quite a lot of exceptions and special cases, but discussion of them is postponed until later sections.

4.1. Introductory Examples

In this section we give basic examples of using BLL lambda expressions in STL algorithm invocations. We start with some simple expressions and work up. First, we initialize the elements of a container, say, a `list`, to the value `1`:

```
list<int> v(10);
for_each(v.begin(), v.end(), _1 = 1);
```

The expression `_1 = 1` creates a lambda functor which assigns the value `1` to every element in `v`.[1]

Next, we create a container of pointers and make them point to the elements in the first container `v`:

```
vector<int*> vp(10);
transform(v.begin(), v.end(), vp.begin(), &_1);
```

The expression `&_1` creates a function object for getting the address of each element in `v`. The addresses get assigned to the corresponding elements in `vp`.

The next code fragment changes the values in `v`. For each element, the function `foo` is called. The original value of the element is passed as an argument to `foo`. The result of `foo` is assigned back to the element:

```
int foo(int);
for_each(v.begin(), v.end(), _1 = bind(foo, _1));
```

The next step is to sort the elements of `vp`:

```
sort(vp.begin(), vp.end(), *_1 > *_2);
```

In this call to `sort`, we are sorting the elements by their contents in descending order.

Finally, the following `for_each` call outputs the sorted content of `vp` separated by line breaks:

```
for_each(vp.begin(), vp.end(), cout << *_1 << '\n');
```

Note that a normal (non−lambda) expression as subexpression of a lambda expression is evaluated immediately. This may cause surprises. For instance, if the previous example is rewritten as

```
for_each(vp.begin(), vp.end(), cout << '\n' << *_1);
```

the subexpression `cout << '\n'` is evaluated immediately and the effect is to output a single line break, followed by the elements of `vp`. The BLL provides functions `constant` and `var` to turn constants and, respectively, variables into lambda expressions, and can be used to prevent the immediate evaluation of subexpressions:

```
for_each(vp.begin(), vp.end(), cout << constant('\n') << *_1);
```

These functions are described more thoroughly in Section 5.5, Delaying constants and variables

4.2. Parameter and return types of lambda functors

During the invocation of a lambda functor, the actual arguments are substituted for the placeholders. The placeholders do not dictate the type of these actual arguments. The basic rule is that a lambda function can be called with arguments of any types, as long as the lambda expression with substitutions performed is a valid C++ expression. As an example, the expression `_1 + _2` creates a binary lambda functor. It can be called with two objects of any types `A` and `B` for which `operator+(A,B)` is defined (and for which BLL knows the return type of the operator, see below).

C++ lacks a mechanism to query a type of an expression. However, this precise mechanism is crucial for the implementation of C++ lambda expressions. Consequently, BLL includes a somewhat complex type deduction system which uses a set of traits classes for deducing the resulting type of lambda functions. It handles expressions where the operands are of built−in types and many of the expressions with operands of standard library types. Many of the user defined types are covered as well, particularly if the user defined operators obey normal conventions in defining the return types.

There are, however, cases when the return type cannot be deduced. For example, suppose you have defined:

```
C operator+(A, B);
```

The following lambda function invocation fails, since the return type cannot be deduced:

```
A a; B b; (_1 + _2)(a, b);
```

There are two alternative solutions to this. The first is to extend the BLL type deduction system to cover your own types (see Section 6, Extending return type deduction system). The second is to use a special lambda expression (`ret`) which defines the return type in place (see Section 5.4, Overriding the deduced return type):

```
A a; B b; ret<C>(_1 + _2)(a, b);
```

For bind expressions, the return type can be defined as a template argument of the bind function as well:

```
bind<int>(foo, _1, _2);
```

4.3. About actual arguments to lambda functors

A general restriction for the actual arguments is that they cannot be non−const rvalues. For example:

```
int i = 1; int j = 2;
(_1 + _2)(i, j); // ok
(_1 + _2)(1, 2); // error (!)
```

This restriction is not as bad as it may look. Since the lambda functors are most often called inside STL−algorithms, the arguments originate from dereferencing iterators and the dereferencing operators seldom return rvalues. And for the cases where they do, there are workarounds discussed in Section 5.9.2, Rvalues as actual arguments to lambda functors .

4.4. Storing bound arguments in lambda functions

By default, temporary const copies of the bound arguments are stored in the lambda functor. This means that the value of a bound argument is fixed at the time of the creation of the lambda function and remains constant during the lifetime of the lambda function object. For example:

```
int i = 1;
(_1 = 2, _1 + i)(i);
```

The comma operator is overloaded to combine lambda expressions into a sequence; the resulting unary lambda functor first assigns 2 to its argument, then adds the value of `i` to it. The value of the expression in the last line is 3, not 4. In other words, the lambda expression that is created is `lambda x.(x = 2, x + 1)` rather than `lambda x.(x = 2, x + i)`.

As said, this is the default behavior for which there are exceptions. The exact rules are as follows:

- The programmer can control the storing mechanism with `ref` and `cref` wrappers [ref]. Wrapping an argument with `ref`, or `cref`, instructs the library to store the argument as a reference, or as a reference to const respectively. For example, if we rewrite the previous example and wrap the variable `i` with `ref`, we are creating the lambda expression `lambda x.(x = 2, x + i)` and the value of the expression in the last line will be 4:

  ```
  i = 1;
  (_1 = 2, _1 + ref(i))(i);
  ```

  Note that `ref` and `cref` are different from `var` and `constant`. While the latter ones create lambda functors, the former do not. For example:

  ```
  int i;
  var(i) = 1; // ok
  ref(i) = 1; // not ok, ref(i) is not a lambda functor
  ```

  The functions `ref` and `cref` mostly exist for historical reasons, and `ref` can always be replaced with `var`, and `cref` with `constant_ref`. See Section 5.5, Delaying constants and variables for details. The `ref` and `cref` functions are general purpose utility functions in Boost, and hence defined directly in the `boost` namespace.
- Array types cannot be copied, they are thus stored as const reference by default.
- For some expressions it makes more sense to store the arguments as references. For example, the obvious intention of the lambda expression `i += _1` is that calls to the lambda functor affect the value of the variable `i`, rather than some temporary copy of it. As another example, the streaming operators take their leftmost argument as non−const references. The exact rules are:

  - The left argument of compound assignment operators (+=, *=, etc.) are stored as references to non−const.
  - If the left argument of << or >> operator is derived from an instantiation of `basic_ostream` or respectively from `basic_istream`, the argument is stored as a reference to non−const. For all other types, the argument is stored as a copy.
  - In pointer arithmetic expressions, non−const array types are stored as non−const references. This is to prevent pointer arithmetic making non−const arrays const.

5. Lambda expressions in details

This section describes different categories of lambda expressions in details. We devote a separate section for each of the possible forms of a lambda expression.

5.1. Placeholders

The BLL defines three placeholder types: `placeholder1_type`, `placeholder2_type` and `placeholder3_type`. BLL has a predefined placeholder variable for each placeholder type: `_1`, `_2` and `_3`. However, the user is not forced to use these placeholders. It is easy to define placeholders with alternative names. This is done by defining new variables of placeholder types. For example:

```
boost::lambda::placeholder1_type X;
boost::lambda::placeholder2_type Y;
boost::lambda::placeholder3_type Z;
```

With these variables defined, `X += Y * Z` is equivalent to `_1 += _2 * _3`.

The use of placeholders in the lambda expression determines whether the resulting function is nullary, unary, binary or 3−ary. The highest placeholder index is decisive. For example:

```
_1 + 5              // unary
_1 * _1 + _1        // unary
_1 + _2             // binary
bind(f, _1, _2, _3) // 3-ary
_3 + 10             // 3-ary
```

Note that the last line creates a 3−ary function, which adds `10` to its *third* argument. The first two arguments are discarded. Furthermore, lambda functors only have a minimum arity. One can always provide more arguments (up the number of supported placeholders) that is really needed. The remaining arguments are just discarded. For example:

```
int i, j, k;
_1(i, j, k)        // returns i, discards j and k
(_2 + _2)(i, j, k) // returns j+j, discards i and k
```

See Section 1, Lambda functor arity for the design rationale behind this functionality.

In addition to these three placeholder types, there is also a fourth placeholder type `placeholderE_type`. The use of this placeholder is defined in Section 5.7, Exceptions describing exception handling in lambda expressions.

When an actual argument is supplied for a placeholder, the parameter passing mode is always by reference. This means that any side−effects to the placeholder are reflected to the actual argument. For example:

```
int i = 1;
(_1 += 2)(i);         // i is now 3
(++_1, cout << _1)(i) // i is now 4, outputs 4
```

5.2. Operator expressions

The basic rule is that any C++ operator invocation with at least one argument being a lambda expression is itself a lambda expression. Almost all overloadable operators are supported. For example, the following is a valid lambda expression:

```
cout << _1, _2[_3] = _1 && false
```

However, there are some restrictions that originate from the C++ operator overloading rules, and some special cases.

5.2.1. Operators that cannot be overloaded

Some operators cannot be overloaded at all (`::`, `.`, `.*`). For some operators, the requirements on return types prevent them to be overloaded to create lambda functors. These operators are `->.`, `->`, `new`, `new[]`, `delete`, `delete[]` and `?:` (the conditional operator).

5.2.2. Assignment and subscript operators

These operators must be implemented as class members. Consequently, the left operand must be a lambda expression. For example:

```
int i;
_1 = i;       // ok
i = _1;       // not ok. i is not a lambda expression
```

There is a simple solution around this limitation, described in Section 5.5, Delaying constants and variables . In short, the left hand argument can be explicitly turned into a lambda functor by wrapping it with a special `var` function:

```
var(i) = _1; // ok
```

5.2.3. Logical operators

Logical operators obey the short−circuiting evaluation rules. For example, in the following code, `i` is never incremented:

```
bool flag = true; int i = 0;
(_1 || ++_2)(flag, i);
```

5.2.4. Comma operator

Comma operator is the statement separator in lambda expressions. Since comma is also the separator between arguments in a function call, extra parenthesis are sometimes needed:

```
for_each(a.begin(), a.end(), (++_1, cout << _1));
```

Without the extra parenthesis around `++_1, cout << _1`, the code would be interpreted as an attempt to call `for_each` with four arguments.

The lambda functor created by the comma operator adheres to the C++ rule of always evaluating the left operand before the right one. In the above example, each element of `a` is first incremented, then written to the stream.

5.2.5. Function call operator

The function call operators have the effect of evaluating the lambda functor. Calls with too few arguments lead to a compile time error.

5.2.6. Member pointer operator

The member pointer operator `operator->*` can be overloaded freely. Hence, for user defined types, member pointer operator is no special case. The built−in meaning, however, is a somewhat more complicated case. The built−in member pointer operator is applied if the left argument is a pointer to an object of some class `A`, and the right hand argument is a pointer to a member of `A`, or a pointer to a member of a class from which `A` derives. We must separate two cases:

- The right hand argument is a pointer to a data member. In this case the lambda functor simply performs the argument substitution and calls the built−in member pointer operator, which returns a reference to the member pointed to. For example:

  ```
  struct A { int d; };
  A* a = new A();
  ```

```
  ...
(a ->* &A::d);      // returns a reference to a->d
(_1 ->* &A::d)(a); // likewise
```

- The right hand argument is a pointer to a member function. For a built−in call like this, the result is kind of a delayed member function call. Such an expression must be followed by a function argument list, with which the delayed member function call is performed. For example:

```
struct B { int foo(int); };
B* b = new B();
  ...
(b ->* &B::foo)         // returns a delayed call to b->foo
                        // a function argument list must follow
(b ->* &B::foo)(1)      // ok, calls b->foo(1)

(_1 ->* &B::foo)(b);    // returns a delayed call to b->foo,
                        // no effect as such
(_1 ->* &B::foo)(b)(1); // calls b->foo(1)
```

5.3. Bind expressions

Bind expressions can have two forms:

```
bind(target-function, bind-argument-list)
bind(target-member-function, object-argument, bind-argument-list)
```

A bind expression delays the call of a function. If this *target function* is *n*−ary, then the `bind-argument-list` must contain *n* arguments as well. In the current version of the BLL, 0 <= n <= 9 must hold. For member functions, the number of arguments must be at most 8, as the object argument takes one argument position. Basically, the `bind-argument-list` must be a valid argument list for the target function, except that any argument can be replaced with a placeholder, or more generally, with a lambda expression. Note that also the target function can be a lambda expression. The result of a bind expression is either a nullary, unary, binary or 3−ary function object depending on the use of placeholders in the `bind-argument-list` (see Section 5.1, Placeholders).

The return type of the lambda functor created by the bind expression can be given as an explicitly specified template parameter, as in the following example:

```
bind<RET>(target-function, bind-argument-list)
```

This is only necessary if the return type of the target function cannot be deduced.

The following sections describe the different types of bind expressions.

5.3.1. Function pointers or references as targets

The target function can be a pointer or a reference to a function and it can be either bound or unbound. For example:

```
X foo(A, B, C); A a; B b; C c;
bind(foo, _1, _2, c)(a, b);
bind(&foo, _1, _2, c)(a, b);
bind(_1, a, b, c)(foo);
```

The return type deduction always succeeds with this type of bind expressions.

Note, that in C++ it is possible to take the address of an overloaded function only if the address is assigned to, or used as an initializer of, a variable, the type of which solves the amibiguity, or if an explicit cast expression is used. This means that overloaded functions cannot be used in bind expressions directly, e.g.:

```
void foo(int);
void foo(float);
int i;
```

```
  ...
bind(&foo, _1)(i);                            // error
  ...
void (*pf1)(int) = &foo;
bind(pf1, _1)(i);                             // ok
bind(static_cast<void(*)(int)>(&foo), _1)(i); // ok
```

5.3.2. Member functions as targets

The syntax for using pointers to member function in bind expression is:

bind(*target-member-function*, *object-argument*, *bind-argument-list*)

The object argument can be a reference or pointer to the object, the BLL supports both cases with a uniform interface:

```
bool A::foo(int) const;
A a;
vector<int> ints;
  ...
find_if(ints.begin(), ints.end(), bind(&A::foo, a, _1));
find_if(ints.begin(), ints.end(), bind(&A::foo, &a, _1));
```

Similarly, if the object argument is unbound, the resulting lambda functor can be called both via a pointer or a reference:

```
bool A::foo(int);
list<A> refs;
list<A*> pointers;
  ...
find_if(refs.begin(), refs.end(), bind(&A::foo, _1, 1));
find_if(pointers.begin(), pointers.end(), bind(&A::foo, _1, 1));
```

Even though the interfaces are the same, there are important semantic differences between using a pointer or a reference as the object argument. The differences stem from the way `bind`–functions take their parameters, and how the bound parameters are stored within the lambda functor. The object argument has the same parameter passing and storing mechanism as any other bind argument slot (see Section 4.4, Storing bound arguments in lambda functions); it is passed as a const reference and stored as a const copy in the lambda functor. This creates some asymmetry between the lambda functor and the original member function, and between seemingly similar lambda functors. For example:

```
class A {
  int i; mutable int j;
public:

  A(int ii, int jj) : i(ii), j(jj) {};
  void set_i(int x) { i = x; };
  void set_j(int x) const { j = x; };
};
```

When a pointer is used, the behavior is what the programmer might expect:

```
A a(0,0); int k = 1;
bind(&A::set_i, &a, _1)(k); // a.i == 1
bind(&A::set_j, &a, _1)(k); // a.j == 1
```

Even though a const copy of the object argument is stored, the original object `a` is still modified. This is since the object argument is a pointer, and the pointer is copied, not the object it points to. When we use a reference, the behaviour is different:

```
A a(0,0); int k = 1;
bind(&A::set_i, a, _1)(k); // error; a const copy of a is stored.
                           // Cannot call a non-const function set_i
bind(&A::set_j, a, _1)(k); // a.j == 0, as a copy of a is modified
```

To prevent the copying from taking place, one can use the `ref` or `cref` wrappers (`var` and `constant_ref` would do as well):

```
bind(&A::set_i, ref(a), _1)(k); // a.j == 1
bind(&A::set_j, cref(a), _1)(k); // a.j == 1
```

Note that the preceding discussion is relevant only for bound arguments. If the object argument is unbound, the parameter passing mode is always by reference. Hence, the argument `a` is not copied in the calls to the two lambda functors below:

```
A a(0,0);
bind(&A::set_i, _1, 1)(a); // a.i == 1
bind(&A::set_j, _1, 1)(a); // a.j == 1
```

5.3.3. Member variables as targets

A pointer to a member variable is not really a function, but the first argument to the `bind` function can nevertheless be a pointer to a member variable. Invoking such a bind expression returns a reference to the data member. For example:

```
struct A { int data; };
A a;
bind(&A::data, _1)(a) = 1;     // a.data == 1
```

The cv−qualifiers of the object whose member is accessed are respected. For example, the following tries to write into a const location:

```
const A ca = a;
bind(&A::data, _1)(ca) = 1;     // error
```

5.3.4. Function objects as targets

Function objects, that is, class objects which have the function call operator defined, can be used as target functions. In general, BLL cannot deduce the return type of an arbitrary function object. However, there are two methods for giving BLL this capability for a certain function object class.

The result_type typedef

The BLL supports the standard library convention of declaring the return type of a function object with a member typedef named `result_type` in the function object class. Here is a simple example:

```
struct A {
  typedef B result_type;
  B operator()(X, Y, Z);
};
```

If a function object does not define a `result_type` typedef, the method described below (`sig` template) is attempted to resolve the return type of the function object. If a function object defines both `result_type` and `sig`, `result_type` takes precedence.

The sig template

Another mechanism that make BLL aware of the return type(s) of a function object is defining member template struct `sig<Args>` with a typedef `type` that specifies the return type. Here is a simple example:

```
struct A {
  template <class Args> struct sig { typedef B type; }
  B operator()(X, Y, Z);
};
```

The template argument `Args` is a `tuple` (or more precisely a `cons` list) type [tuple], where the first element is the function object type itself, and the remaining elements are the types of the arguments, with which the function object is being called.

This may seem overly complex compared to defining the `result_type` typedef. Howver, there are two significant restrictions with using just a simple typedef to express the return type:

1. If the function object defines several function call operators, there is no way to specify different result types for them.
2. If the function call operator is a template, the result type may depend on the template parameters. Hence, the typedef ought to be a template too, which the C++ language does not support.

The following code shows an example, where the return type depends on the type of one of the arguments, and how that dependency can be expressed with the `sig` template:

```
struct A {

  // the return type equals the third argument type:
  template<class T1, class T2, class T3>
  T3 operator()(const T1& t1, const T2& t2, const T3& t3) const;

  template <class Args>
  class sig {
    // get the third argument type (4th element)
    typedef typename
      boost::tuples::element<3, Args>::type T3;
  public:
    typedef typename
      boost::remove_cv<T3>::type type;
  };
};
```

The elements of the `Args` tuple are always non−reference types. Moreover, the element types can have a const or volatile qualifier (jointly referred to as *cv−qualifiers*), or both. This is since the cv−qualifiers in the arguments can affect the return type. The reason for including the potentially cv−qualified function object type itself into the `Args` tuple, is that the function object class can contain both const and non−const (or volatile, even const volatile) function call operators, and they can each have a different return type.

The `sig` template can be seen as a *meta−function* that maps the argument type tuple to the result type of the call made with arguments of the types in the tuple. As the example above demonstrates, the template can end up being somewhat complex. Typical tasks to be performed are the extraction of the relevant types from the tuple, removing cv−qualifiers etc. See the Boost type_traits [type_traits] and Tuple [type_traits] libraries for tools that can aid in these tasks. The `sig` templates are a refined version of a similar mechanism first introduced in the FC++ library [fc++].

5.4. Overriding the deduced return type

The return type deduction system may not be able to deduce the return types of some user defined operators or bind expressions with class objects. A special lambda expression type is provided for stating the return type explicitly and overriding the deduction system. To state that the return type of the lambda functor defined by the lambda expression `e` is `T`, you can write:

```
ret<T>(e);
```

The effect is that the return type deduction is not performed for the lambda expression `e` at all, but instead, `T` is used as the return type. Obviously `T` cannot be an arbitrary type, the true result of the lambda functor must be implicitly convertible to `T`. For example:

```
A a; B b;
C operator+(A, B);
int operator*(A, B);
  ...
ret<D>(_1 + _2)(a, b);     // error (C cannot be converted to D)
ret<C>(_1 + _2)(a, b);     // ok
ret<float>(_1 * _2)(a, b); // ok (int can be converted to float)
  ...
```

```
struct X {
  Y operator(int)();
};
  ...
X x; int i;
bind(x, _1)(i);             // error, return type cannot be deduced
ret<Y>(bind(x, _1))(i);     // ok
```

For bind expressions, there is a short−hand notation that can be used instead of `ret`. The last line could alternatively be written as:

```
bind<Z>(x, _1)(i);
```

This feature is modeled after the Boost Bind library [bind].

Note that within nested lambda expressions, the `ret` must be used at each subexpression where the deduction would otherwise fail. For example:

```
A a; B b;
C operator+(A, B); D operator-(C);
  ...
ret<D>( - (_1 + _2))(a, b); // error
ret<D>( - ret<C>(_1 + _2))(a, b); // ok
```

If you find yourself using `ret` repeatedly with the same types, it is worth while extending the return type deduction (see Section 6, Extending return type deduction system).

5.4.1. Nullary lambda functors and ret

As stated above, the effect of `ret` is to prevent the return type deduction to be performed. However, there is an exception. Due to the way the C++ template instantiation works, the compiler is always forced to instantiate the return type deduction templates for zero−argument lambda functors. This introduces a slight problem with `ret`, best described with an example:

```
struct F { int operator()(int i) const; };
F f;
  ...
bind(f, _1);           // fails, cannot deduce the return type
ret<int>(bind(f, _1)); // ok
  ...
bind(f, 1);            // fails, cannot deduce the return type
ret<int>(bind(f, 1));  // fails as well!
```

The BLL cannot deduce the return types of the above bind calls, as `F` does not define the typedef `result_type`. One would expect `ret` to fix this, but for the nullary lambda functor that results from a bind expression (last line above) this does not work. The return type deduction templates are instantiated, even though it would not be necessary and the result is a compilation error.

The solution to this is not to use the `ret` function, but rather define the return type as an explicitly specified template parameter in the `bind` call:

```
bind<int>(f, 1);       // ok
```

The lambda functors created with `ret<`*`T`*`>(bind(`*`arg-list`*`))` and `bind<`*`T`*`>(`*`arg-list`*`)` have the exact same functionality apart from the fact that for some nullary lambda functors the former does not work while the latter does.

5.5. Delaying constants and variables

The unary functions `constant`, `constant_ref` and `var` turn their argument into a lambda functor, that implements an identity mapping. The former two are for constants, the latter for variables. The use of these *delayed* constants and variables is sometimes necessary due to the lack of explicit syntax for lambda expressions. For example:

```
for_each(a.begin(), a.end(), cout << _1 << ' ');
for_each(a.begin(), a.end(), cout << ' ' << _1);
```

The first line outputs the elements of `a` separated by spaces, while the second line outputs a space followed by the elements of `a` without any separators. The reason for this is that neither of the operands of `cout << ' '` is a lambda expression, hence `cout << ' '` is evaluated immediately. To delay the evaluation of `cout << ' '`, one of the operands must be explicitly marked as a lambda expression. This is accomplished with the `constant` function:

```
for_each(a.begin(), a.end(), cout << constant(' ') << _1);
```

The call `constant(' ')` creates a nullary lambda functor which stores the character constant `' '` and returns a reference to it when invoked. The function `constant_ref` is similar, except that it stores a constant reference to its argument. The `constant` and `consant_ref` are only needed when the operator call has side effects, like in the above example.

Sometimes we need to delay the evaluation of a variable. Suppose we wanted to output the elements of a container in a numbered list:

```
int index = 0;
for_each(a.begin(), a.end(), cout << ++index << ':' << _1 << '\n');
for_each(a.begin(), a.end(), cout << ++var(index) << ':' << _1 << '\n');
```

The first `for_each` invocation does not do what we want; `index` is incremented only once, and its value is written into the output stream only once. By using `var` to make `index` a lambda expression, we get the desired effect.

In sum, `var(x)` creates a nullary lambda functor, which stores a reference to the variable `x`. When the lambda functor is invoked, a reference to `x` is returned.

Naming delayed constants and variables

It is possible to predefine and name a delayed variable or constant outside a lambda expression. The templates `var_type`, `constant_type` and `constant_ref_type` serve for this purpose. They are used as:

```
var_type<T>::type delayed_i(var(i));
constant_type<T>::type delayed_c(constant(c));
```

The first line defines the variable `delayed_i` which is a delayed version of the variable `i` of type `T`. Analogously, the second line defines the constant `delayed_c` as a delayed version of the constant `c`. For example:

```
int i = 0; int j;
for_each(a.begin(), a.end(), (var(j) = _1, _1 = var(i), var(i) = var(j)));
```

is equivalent to:

```
int i = 0; int j;
var_type<int>::type vi(var(i)), vj(var(j));
for_each(a.begin(), a.end(), (vj = _1, _1 = vi, vi = vj));
```

Here is an example of naming a delayed constant:

```
constant_type<char>::type space(constant(' '));
for_each(a.begin(),a.end(), cout << space << _1);
```

About assignment and subscript operators

As described in Section 5.2.2, Assignment and subscript operators , assignment and subscripting operators are always defined as member functions. This means, that for expressions of the form `x = y` or `x[y]` to be interpreted as lambda expressions, the left−hand operand `x` must be a lambda expression. Consequently, it is sometimes necessary to use `var` for this purpose. We repeat the example from Section 5.2.2, Assignment and subscript operators :

```
int i;
i = _1;        // error
var(i) = _1;   // ok
```

Note that the compound assignment operators `+=`, `-=` etc. can be defined as non−member functions, and thus they are interpreted as lambda expressions even if only the right−hand operand is a lambda expression. Nevertheless, it is perfectly ok to delay the left operand explicitly. For example, `i += _1` is equivalent to `var(i) += _1`.

5.6. Lambda expressions for control structures

BLL defines several functions to create lambda functors that represent control structures. They all take lambda functors as parameters and return `void`. To start with an example, the following code outputs all even elements of some container `a`:

```
for_each(a.begin(), a.end(),
         if_then(_1 % 2 == 0, cout << _1));
```

The BLL supports the following function templates for control structures:

```
if_then(condition, then_part)
if_then_else(condition, then_part, else_part)
if_then_else_return(condition, then_part, else_part)
while_loop(condition, body)
while_loop(condition) // no body case
do_while_loop(condition, body)
do_while_loop(condition) // no body case
for_loop(init, condition, increment, body)
for_loop(init, condition, increment) // no body case
switch_statement(...)
```

The return types of all control construct lambda functor is `void`, except for `if_then_else_return`, which wraps a call to the conditional operator

```
condition ? then_part : else_part
```

The return type rules for this operator are somewhat complex. Basically, if the branches have the same type, this type is the return type. If the type of the branches differ, one branch, say of type `A`, must be convertible to the other branch, say of type `B`. In this situation, the result type is `B`. Further, if the common type is an lvalue, the return type will be an lvalue too.

Delayed variables tend to be commonplace in control structure lambda expressions. For instance, here we use the `var` function to turn the arguments of `for_loop` into lambda expressions. The effect of the code is to add 1 to each element of a two−dimensional array:

```
int a[5][10]; int i;
for_each(a, a+5,
  for_loop(var(i)=0, var(i)<10, ++var(i),
           _1[var(i)] += 1));
```

The BLL supports an alternative syntax for control expressions, suggested by Joel de Guzmann. By overloading the `operator[]` we can get a closer resemblance with the built−in control structures:

```
if_(condition)[then_part]
if_(condition)[then_part].else_[else_part]
while_(condition)[body]
do_[body].while_(condition)
for_(init, condition, increment)[body]
```

For example, using this syntax the `if_then` example above can be written as:

```
for_each(a.begin(), a.end(),
         if_(_1 % 2 == 0)[ cout << _1 ])
```

As more experience is gained, we may end up deprecating one or the other of these syntaces.

5.6.1. Switch statement

The lambda expressions for `switch` control structures are more complex since the number of cases may vary. The general form of a switch lambda expression is:

```
switch_statement(condition,
  case_statement<label>(lambda expression),
  case_statement<label>(lambda expression),
  ...
  default_statement(lambda expression)
)
```

The *`condition`* argument must be a lambda expression that creates a lambda functor with an integral return type. The different cases are created with the `case_statement` functions, and the optional default case with the `default_statement` function. The case labels are given as explicitly specified template arguments to `case_statement` functions and `break` statements are implicitly part of each case. For example, `case_statement<1>(a)`, where `a` is some lambda functor, generates the code:

```
case 1:
  evaluate lambda functor a;
  break;
```

The `switch_statement` function is specialized for up to 9 case statements.

As a concrete example, the following code iterates over some container `v` and ouptuts zero for each `0`, one for each `1`, and other: *`n`* for any other value *n*. Note that another lambda expression is sequenced after the `switch_statement` to output a line break after each element:

```
std::for_each(v.begin(), v.end(),
  (
    switch_statement(
      _1,
      case_statement<0>(std::cout << constant("zero")),
      case_statement<1>(std::cout << constant("one")),
      default_statement(cout << constant("other: ") << _1)
    ),
    cout << constant("\n")
  )
);
```

5.7. Exceptions

The BLL provides lambda functors that throw and catch exceptions. Lambda functors for throwing exceptions are created with the unary function `throw_exception`. The argument to this function is the exception to be thrown, or a lambda functor which creates the exception to be thrown. A lambda functor for rethrowing exceptions is created with the nullary `rethrow` function.

Lambda expressions for handling exceptions are somewhat more complex. The general form of a lambda expression for try catch blocks is as follows:

```
try_catch(
  lambda expression,
  catch_exception<type>(lambda expression),
  catch_exception<type>(lambda expression),
  ...
  catch_all(lambda expression)
)
```

The first lambda expression is the try block. Each `catch_exception` defines a catch block where the explicitly specified template argument defines the type of the exception to catch. The lambda expression within the `catch_exception` defines the actions to take if the exception is caught. Note that the resulting exception handlers catch the exceptions as references, i.e., `catch_exception<T>(...)` results in the catch block:

```
catch(T& e) { ... }
```

The last catch block can alternatively be a call to `catch_exception<`*`type`*`>` or to `catch_all`, which is the lambda expression equivalent to `catch(...)`.

The Example 4.1, Throwing and handling exceptions in lambda expressions. demonstrates the use of the BLL exception handling tools. The first handler catches exceptions of type `foo_exception`. Note the use of `_1` placeholder in the body of the handler.

The second handler shows how to throw exceptions, and demonstrates the use of the *exception placeholder* `_e`. It is a special placeholder, which refers to the caught exception object within the handler body. Here we are handling an exception of type `std::exception`, which carries a string explaining the cause of the exception. This explanation can be queried with the zero−argument member function `what`. The expression `bind(&std::exception::what, _e)` creates the lambda function for making that call. Note that `_e` cannot be used outside of an exception handler lambda expression. The last line of the second handler constructs a new exception object and throws that with `throw exception`. Constructing and destructing objects within lambda expressions is explained in Section 5.8, Construction and destruction

Finally, the third handler (`catch_all`) demonstrates rethrowing exceptions.

Example 4.1. Throwing and handling exceptions in lambda expressions.

```
for_each(
  a.begin(), a.end(),
  try_catch(
    bind(foo, _1),                 // foo may throw
    catch_exception<foo_exception>(
      cout << constant("Caught foo_exception: ")
           << "foo was called with argument = " << _1
    ),
    catch_exception<std::exception>(
      cout << constant("Caught std::exception: ")
           << bind(&std::exception::what, _e),
      throw_exception(bind(constructor<bar_exception>(), _1)))
    ),
    catch_all(
      (cout << constant("Unknown"), rethrow())
    )
  )
);
```

5.8. Construction and destruction

Operators `new` and `delete` can be overloaded, but their return types are fixed. Particularly, the return types cannot be lambda functors, which prevents them to be overloaded for lambda expressions. It is not possible to take the address of a constructor, hence constructors cannot be used as target functions in bind expressions. The same is true for destructors. As a way around these constraints, BLL defines wrapper classes for `new` and `delete` calls, as well as for constructors and destructors. Instances of these classes are function objects, that can be used as target functions of bind expressions. For example:

```
int* a[10];
for_each(a, a+10, _1 = bind(new_ptr<int>()));
for_each(a, a+10, bind(delete_ptr(), _1));
```

The `new_ptr<int>()` expression creates a function object that calls `new int()` when invoked, and wrapping that inside `bind` makes it a lambda functor. In the same way, the expression `delete_ptr()` creates a function object that invokes `delete` on its argument. Note that `new_ptr<T>()` can take arguments as well. They are passed directly to the constructor invocation and thus allow calls to constructors which take arguments.

As an example of constructor calls in lambda expressions, the following code reads integers from two containers `x` and `y`, constructs pairs out of them and inserts them into a third container:

```
vector<pair<int, int> > v;
transform(x.begin(), x.end(), y.begin(), back_inserter(v),
          bind(constructor<pair<int, int> >(), _1, _2));
```

Table 4.1, Construction and destruction related function objects. lists all the function objects related to creating and destroying objects, showing the expression to create and call the function object, and the effect of evaluating that expression.

Table 4.1. Construction and destruction related function objects.

Function object call	Wrapped expression
`constructor<T>()(`*`arg_list`*`)`	T(*`arg_list`*)
`destructor()(a)`	`a.~A()`, where `a` is of type `A`
`destructor()(pa)`	`pa->~A()`, where `pa` is of type `A*`
`new_ptr<T>()(`*`arg_list`*`)`	`new T(`*`arg_list`*`)`
`new_array<T>()(sz)`	`new T[sz]`
`delete_ptr()(p)`	`delete p`
`delete_array()(p)`	`delete p[]`

5.9. Special lambda expressions

5.9.1. Preventing argument substitution

When a lambda functor is called, the default behavior is to substitute the actual arguments for the placeholders within all subexpressions. This section describes the tools to prevent the substitution and evaluation of a subexpression, and explains when these tools should be used.

The arguments to a bind expression can be arbitrary lambda expressions, e.g., other bind expressions. For example:

```
int foo(int); int bar(int);
...
int i;
bind(foo, bind(bar, _1))(i);
```

The last line makes the call `foo(bar(i));` Note that the first argument in a bind expression, the target function, is no exception, and can thus be a bind expression too. The innermost lambda functor just has to return something that can be used as a target function: another lambda functor, function pointer, pointer to member function etc. For example, in the following code the innermost lambda functor makes a selection between two functions, and returns a pointer to one of them:

```
int add(int a, int b) { return a+b; }
int mul(int a, int b) { return a*b; }

int(*)(int, int)  add_or_mul(bool x) {
  return x ? add : mul;
}
```

```
bool condition; int i; int j;
...
bind(bind(&add_or_mul, _1), _2, _3)(condition, i, j);
```

5.9.1.1. Unlambda

A nested bind expression may occur inadvertently, if the target function is a variable with a type that depends on a template parameter. Typically the target function could be a formal parameter of a function template. In such a case, the programmer may not know whether the target function is a lambda functor or not.

Consider the following function template:

```
template<class F>
int nested(const F& f) {
  int x;
  ...
  bind(f, _1)(x);
  ...
}
```

Somewhere inside the function the formal parameter `f` is used as a target function in a bind expression. In order for this `bind` call to be valid, `f` must be a unary function. Suppose the following two calls to `nested` are made:

```
int foo(int);
int bar(int, int);
nested(&foo);
nested(bind(bar, 1, _1));
```

Both are unary functions, or function objects, with appropriate argument and return types, but the latter will not compile. In the latter call, the bind expression inside `nested` will become:

```
bind(bind(bar, 1, _1), _1)
```

When this is invoked with `x`, after substituitions we end up trying to call

```
bar(1, x)(x)
```

which is an error. The call to `bar` returns int, not a unary function or function object.

In the example above, the intent of the bind expression in the `nested` function is to treat `f` as an ordinary function object, instead of a lambda functor. The BLL provides the function template `unlambda` to express this: a lambda functor wrapped inside `unlambda` is not a lambda functor anymore, and does not take part into the argument substitution process. Note that for all other argument types `unlambda` is an identity operation, except for making non−const objects const.

Using `unlambda`, the `nested` function is written as:

```
template<class F>
int nested(const F& f) {
  int x;
  ...
  bind(unlambda(f), _1)(x);
  ...
}
```

5.9.1.2. Protect

The `protect` function is related to unlambda. It is also used to prevent the argument substitution taking place, but whereas `unlambda` turns a lambda functor into an ordinary function object for good, `protect` does this temporarily, for just one evaluation round. For example:

```
int x = 1, y = 10;
(_1 + protect(_1 + 2))(x)(y);
```

The first call substitutes `x` for the leftmost `_1`, and results in another lambda functor `x + (_1 + 2)`, which after the call with `y` becomes `x + (y + 2)`, and thus finally 13.

Primary motivation for including `protect` into the library, was to allow nested STL algorithm invocations (Section 5.11, Nesting STL algorithm invocations).

5.9.2. Rvalues as actual arguments to lambda functors

Actual arguments to the lambda functors cannot be non−const rvalues. This is due to a deliberate design decision: either we have this restriction, or there can be no side−effects to the actual arguments. There are ways around this limitation. We repeat the example from section Section 4.3, About actual arguments to lambda functors and list the different solutions:

```
int i = 1; int j = 2;
(_1 + _2)(i, j); // ok
(_1 + _2)(1, 2); // error (!)
```

1. If the rvalue is of a class type, the return type of the function that creates the rvalue should be defined as const. Due to an unfortunate language restriction this does not work for built−in types, as built−in rvalues cannot be const qualified.
2. If the lambda function call is accessible, the `make_const` function can be used to *constify* the rvalue. E.g.:

   ```
   (_1 + _2)(make_const(1), make_const(2)); // ok
   ```

   Commonly the lambda function call site is inside a standard algorithm function template, preventing this solution to be used.
3. If neither of the above is possible, the lambda expression can be wrapped in a `const_parameters` function. It creates another type of lambda functor, which takes its arguments as const references. For example:

   ```
   const_parameters(_1 + _2)(1, 2); // ok
   ```

   Note that `const_parameters` makes all arguments const. Hence, in the case were one of the arguments is a non−const rvalue, and another argument needs to be passed as a non−const reference, this approach cannot be used.
4. If none of the above is possible, there is still one solution, which unfortunately can break const correctness. The solution is yet another lambda functor wrapper, which we have named `break_const` to alert the user of the potential dangers of this function. The `break_const` function creates a lambda functor that takes its arguments as const, and casts away constness prior to the call to the original wrapped lambda functor. For example:

   ```
   int i;
   ...
   (_1 += _2)(i, 2);                 // error, 2 is a non-const rvalue
   const_parameters(_1 += _2)(i, 2); // error, i becomes const
   break_const(_1 += _2)(i, 2);      // ok, but dangerous
   ```

   Note, that the results of `break_const` or `const_parameters` are not lambda functors, so they cannot be used as subexpressions of lambda expressions. For instance:

   ```
   break_const(_1 + _2) + _3; // fails.
   const_parameters(_1 + _2) + _3; // fails.
   ```

   However, this kind of code should never be necessary, since calls to sub lambda functors are made inside the BLL, and are not affected by the non−const rvalue problem.

5.10. Casts, sizeof and typeid

5.10.1. Cast expressions

The BLL defines its counterparts for the four cast expressions `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`. The BLL versions of the cast expressions have the prefix `ll_`. The type to cast to is given as an explicitly specified template argument, and the sole argument is the expression from which to perform the cast. If the argument is a lambda functor, the lambda functor is evaluated first. For example, the following code uses `ll_dynamic_cast` to count the number of `derived` instances in the container `a`:

```
class base {};
class derived : public base {};

vector<base*> a;
...
int count = 0;
for_each(a.begin(), a.end(),
         if_then(ll_dynamic_cast<derived*>(_1), ++var(count)));
```

5.10.2. Sizeof and typeid

The BLL counterparts for these expressions are named `ll_sizeof` and `ll_typeid`. Both take one argument, which can be a lambda expression. The lambda functor created wraps the `sizeof` or `typeid` call, and when the lambda functor is called the wrapped operation is performed. For example:

```
vector<base*> a;
...
for_each(a.begin(), a.end(),
         cout << bind(&type_info::name, ll_typeid(*_1)));
```

Here `ll_typeid` creates a lambda functor for calling `typeid` for each element. The result of a `typeid` call is an instance of the `type_info` class, and the bind expression creates a lambda functor for calling the `name` member function of that class.

5.11. Nesting STL algorithm invocations

The BLL defines common STL algorithms as function object classes, instances of which can be used as target functions in bind expressions. For example, the following code iterates over the elements of a two–dimensional array, and computes their sum.

```
int a[100][200];
int sum = 0;

std::for_each(a, a + 100,
              bind(ll::for_each(), _1, _1 + 200, protect(sum += _1)));
```

The BLL versions of the STL algorithms are classes, which define the function call operator (or several overloaded ones) to call the corresponding function templates in the `std` namespace. All these structs are placed in the subnamespace `boost::lambda:ll`.

Note that there is no easy way to express an overloaded member function call in a lambda expression. This limits the usefulness of nested STL algorithms, as for instance the `begin` function has more than one overloaded definitions in container templates. In general, something analogous to the pseudo–code below cannot be written:

```
std::for_each(a.begin(), a.end(),
              bind(ll::for_each(), _1.begin(), _1.end(), protect(sum += _1)));
```

Some aid for common special cases can be provided though. The BLL defines two helper function object classes, `call_begin` and `call_end`, which wrap a call to the `begin` and, respectively, `end` functions of a container, and return the `const_iterator` type of the container. With these helper templates, the above code becomes:

```
std::for_each(a.begin(), a.end(),
          bind(ll::for_each(),
               bind(call_begin(), _1), bind(call_end(), _1),
                    protect(sum += _1)));
```

6. Extending return type deduction system

In this section, we explain how to extend the return type deduction system to cover user defined operators. In many cases this is not necessary, as the BLL defines default return types for operators. For example, the default return type for all comparison operators is `bool`, and as long as the user defined comparison operators have a bool return type, there is no need to write new specializations for the return type deduction classes. Sometimes this cannot be avoided, though.

The overloadable user defined operators are either unary or binary. For each arity, there are two traits templates that define the return types of the different operators. Hence, the return type system can be extended by providing more specializations for these templates. The templates for unary functors are `plain_return_type_1<Action, A>` and `return_type_1<Action, A>` , and `plain_return_type_2<Action, A, B>` and `return_type_2<Action, A, B>` respectively for binary functors.

The first parameter (`Action`) to all these templates is the *action* class, which specifies the operator. Operators with similar return type rules are grouped together into *action groups*, and only the action class and action group together define the operator unambiguously. As an example, the action type `arithmetic_action<plus_action>` stands for `operator+`. The complete listing of different action types is shown in Table 4.2, Action types .

The latter parameters, `A` in the unary case, or `A` and `B` in the binary case, stand for the argument types of the operator call. The two sets of templates, `plain_return_type_`*n* and `return_type_`*n* (*n* is 1 or 2) differ in the way how parameter types are presented to them. For the former templates, the parameter types are always provided as non−reference types, and do not have const or volatile qualifiers. This makes specializing easy, as commonly one specialization for each user defined operator, or operator group, is enough. On the other hand, if a particular operator is overloaded for different cv−qualifications of the same argument types, and the return types of these overloaded versions differ, a more fine−grained control is needed. Hence, for the latter templates, the parameter types preserve the cv−qualifiers, and are non−reference types as well. The downside is, that for an overloaded set of operators of the kind described above, one may end up needing up to 16 `return_type_2` specializations.

Suppose the user has overloaded the following operators for some user defined types `X`, `Y` and `Z`:

```
Z operator+(const X&, const Y&);
Z operator-(const X&, const Y&);
```

Now, one can add a specialization stating, that if the left hand argument is of type `X`, and the right hand one of type `Y`, the return type of all such binary arithmetic operators is `Z`:

```
namespace boost {
namespace lambda {

template<class Act>
struct plain_return_type_2<arithmetic_action<Act>, X, Y> {
  typedef Z type;
};

}
}
```

Having this specialization defined, BLL is capable of correctly deducing the return type of the above two operators. Note, that the specializations must be in the same namespace, `::boost::lambda`, with the primary template. For brevity, we do not show the namespace definitions in the examples below.

It is possible to specialize on the level of an individual operator as well, in addition to providing a specialization for a group of operators. Say, we add a new arithmetic operator for argument types `X` and `Y`:

```
X operator*(const X&, const Y&);
```

Our first rule for all arithmetic operators specifies that the return type of this operator is `Z`, which obviously is not the case. Hence, we provide a new rule for the multiplication operator:

```
template<>
struct plain_return_type_2<arithmetic_action<multiply_action>, X, Y> {
  typedef X type;
};
```

The specializations can define arbitrary mappings from the argument types to the return type. Suppose we have some mathematical vector type, templated on the element type:

```
template <class T> class my_vector;
```

Suppose the addition operator is defined between any two `my_vector` instantiations, as long as the addition operator is defined between their element types. Furthermore, the element type of the resulting `my_vector` is the same as the result type of the addition between the element types. E.g., adding `my_vector<int>` and `my_vector<double>` results in `my_vector<double>`. The BLL has traits classes to perform the implicit built–in and standard type conversions between integral, floating point, and complex classes. Using BLL tools, the addition operator described above can be defined as:

```
template<class A, class B>
my_vector<typename return_type_2<arithmetic_action<plus_action>, A, B>::type>
operator+(const my_vector<A>& a, const my_vector<B>& b)
{
  typedef typename
    return_type_2<arithmetic_action<plus_action>, A, B>::type res_type;
  return my_vector<res_type>();
}
```

To allow BLL to deduce the type of `my_vector` additions correctly, we can define:

```
template<class A, class B>
class plain_return_type_2<arithmetic_action<plus_action>,
                           my_vector<A>, my_vector<B> > {
  typedef typename
    return_type_2<arithmetic_action<plus_action>, A, B>::type res_type;
public:
  typedef my_vector<res_type> type;
};
```

Note, that we are reusing the existing specializations for the BLL `return_type_2` template, which require that the argument types are references.

Table 4.2. Action types

`+`	`arithmetic_action<plus_action>`
`-`	`arithmetic_action<minus_action>`
`*`	`arithmetic_action<multiply_action>`
`/`	`arithmetic_action<divide_action>`
`%`	`arithmetic_action<remainder_action>`
`+`	`unary_arithmetic_action<plus_action>`
`-`	`unary_arithmetic_action<minus_action>`
`&`	`bitwise_action<and_action>`

|	bitwise_action<or_action>
~	bitwise_action<not_action>
^	bitwise_action<xor_action>
<<	bitwise_action<leftshift_action_no_stream>
>>	bitwise_action<rightshift_action_no_stream>
&&	logical_action<and_action>
||	logical_action<or_action>
!	logical_action<not_action>
<	relational_action<less_action>
>	relational_action<greater_action>
<=	relational_action<lessorequal_action>
>=	relational_action<greaterorequal_action>
==	relational_action<equal_action>
!=	relational_action<notequal_action>
+=	arithmetic_assignment_action<plus_action>
-=	arithmetic_assignment_action<minus_action>
*=	arithmetic_assignment_action<multiply_action>
/=	arithmetic_assignment_action<divide_action>
%=	arithmetic_assignment_action<remainder_action>
&=	bitwise_assignment_action<and_action>
=|	bitwise_assignment_action<or_action>
^=	bitwise_assignment_action<xor_action>
<<=	bitwise_assignment_action<leftshift_action>
>>=	bitwise_assignment_action<rightshift_action>
++	pre_increment_decrement_action<increment_action>
--	pre_increment_decrement_action<decrement_action>
++	post_increment_decrement_action<increment_action>
--	post_increment_decrement_action<decrement_action>
&	other_action<address_of_action>
*	other_action<contents_of_action>
,	other_action<comma_action>

7. Practical considerations

7.1. Performance

In theory, all overhead of using STL algorithms and lambda functors compared to hand written loops can be optimized away, just as the overhead from standard STL function objects and binders can. Depending on the compiler, this can also be true in practice. We ran two tests with the GCC 3.0.4 compiler on 1.5 GHz Intel Pentium 4. The optimization flag −03 was used.

In the first test we compared lambda functors against explicitly written function objects. We used both of these styles to define unary functions which multiply the argument repeatedly by itself. We started with the identity function, going up to x^5. The expressions were called inside a `std::transform` loop, reading the argument from one `std::vector<int>` and placing the result into another. The length of the vectors was 100 elements. The running times are listed in Table 4.3, Test 1CPU time of expressions with integer multiplication written as a lambda expression and as a traditional hand−coded function object class. The running times are expressed in arbitrary units. . We can observe that there is no significant difference between the two approaches.

In the second test we again used `std::transform` to perform an operation to each element in a 100−element long vector. This time the element type of the vectors was `double` and we started with very simple arithmetic expressions and moved to more complex ones. The running times are listed in Table 4.4, Test 2CPU time of arithmetic expressions written as lambda expressions, as classic STL unnamed functions (using `compose2`, `bind1st` etc.) and as traditional hand−coded function object classes. Using BLL terminology, `a` and `b` are bound arguments in the expressions, and `x` is open. All variables were of types `double`. The running times are expressed in arbitrary units. . Here, we also included classic STL style unnamed functions into tests. We do not show these expressions, as they get rather complex. For example, the last expression in Table 4.4, Test 2CPU time of arithmetic expressions written as lambda expressions, as classic STL unnamed functions (using `compose2`, `bind1st` etc.) and as traditional hand−coded function object classes. Using BLL terminology, `a` and `b` are bound arguments in the expressions, and `x` is open. All variables were of types `double`. The running times are expressed in arbitrary units. written with classic STL tools contains 7 calls to `compose2`, 8 calls to `bind1st` and altogether 14 constructor invocations for creating `multiplies`, `minus` and `plus` objects. In this test the BLL expressions are a little slower (roughly 10% on average, less than 14% in all cases) than the corresponding hand−written function objects. The performance hit is a bit greater with classic STL expressions, up to 27% for the simplest expressios.

The tests suggest that the BLL does not introduce a loss of performance compared to STL function objects. With a reasonable optimizing compiler, one should expect the performance characteristics be comparable to using classic STL. Moreover, with simple expressions the performance can be expected to be close to that of explicitly written function objects. Note however, that evaluating a lambda functor consist of a sequence of calls to small functions that are declared inline. If the compiler fails to actually expand these functions inline, the performance can suffer. The running time can more than double if this happens. Although the above tests do not include such an expression, we have experienced this for some seemingly simple expressions.

Table 4.3. Test 1CPU time of expressions with integer multiplication written as a lambda expression and as a traditional hand−coded function object class. The running times are expressed in arbitrary units.

CPU time of expressions with integer multiplication written as a lambda expression and as a traditional hand−coded function object class. The running times are expressed in arbitrary units.

expression	lambda expression	hand−coded function object
x	240	230
x*x	340	350
x*x*x	770	760
x*x*x*x	1180	1210
x*x*x*x*x	1950	1910

Table 4.4. Test 2CPU time of arithmetic expressions written as lambda expressions, as classic STL unnamed functions (using `compose2`, `bind1st` etc.) and as traditional hand–coded function object classes. Using BLL terminology, `a` and `b` are bound arguments in the expressions, and `x` is open. All variables were of types `double`. The running times are expressed in arbitrary units.

CPU time of arithmetic expressions written as lambda expressions, as classic STL unnamed functions (using `compose2`, `bind1st` etc.) and as traditional hand–coded function object classes. Using BLL terminology, `a` and `b` are bound arguments in the expressions, and `x` is open. All variables were of types `double`. The running times are expressed in arbitrary units.

expression	lambda expression	classic STL expression	hand–coded function object
ax	330	370	290
−ax	350	370	310
ax−(a+x)	470	500	420
(ax−(a+x))(a+x)	620	670	600
((ax) − (a+x))(bx − (b+x))(ax − (b+x))(bx − (a+x))	1660	1660	1460

Some additional performance testing with an earlier version of the library is described [Jär00].

7.2. About compiling

The BLL uses templates rather heavily, performing numerous recursive instantiations of the same templates. This has (at least) three implications:

- While it is possible to write incredibly complex lambda expressions, it probably isn't a good idea. Compiling such expressions may end up requiring a lot of memory at compile time, and being slow to compile.
- The types of lambda functors that result from even the simplest lambda expressions are cryptic. Usually the programmer doesn't need to deal with the lambda functor types at all, but in the case of an error in a lambda expression, the compiler usually outputs the types of the lambda functors involved. This can make the error messages very long and difficult to interpret, particularly if the compiler outputs the whole chain of template instantiations.
- The C++ Standard suggests a template nesting level of 17 to help detect infinite recursion. Complex lambda templates can easily exceed this limit. Most compilers allow a greater number of nested templates, but commonly require the limit explicitly increased with a command line argument.

7.3. Portability

The BLL works with the following compilers, that is, the compilers are capable of compiling the test cases that are included with the BLL:

- GCC 3.0.4
- KCC 4.0f with EDG 2.43.1
- GCC 2.96 (fails with one test case, the `exception_test.cpp` results in an internal compiler error.)

7.3.1. Test coverage

The following list describes the test files included and the features that each file covers:

- `bind_tests_simple.cpp` : Bind expressions of different arities and types of target functions: function pointers, function objects and member functions. Function composition with bind expressions.
- `bind_tests_simple_function_references.cpp` : Repeats all tests from `bind_tests_simple.cpp` where the target function is a function pointer, but uses function references instead.

- `bind_tests_advanced.cpp` : Contains tests for nested bind expressions, `unlambda`, `protect`, `const_parameters` and `break_const`. Tests passing lambda functors as actual arguments to other lambda functors, currying, and using the `sig` template to specify the return type of a function object.
- `operator_tests_simple.cpp` : Tests using all operators that are overloaded for lambda expressions, that is, unary and binary arithmetic, bitwise, comparison, logical, increment and decrement, compound, assignment, subscrict, address of, dereference, and comma operators. The streaming nature of shift operators is tested, as well as pointer arithmetic with plus and minus operators.
- `member_pointer_test.cpp` : The pointer to member operator is complex enough to warrant a separate test file.
- `control_structures.cpp` : Tests for the looping and if constructs.
- `switch_construct.cpp` : Includes tests for all supported arities of the switch statement, both with and without the default case.
- `exception_test.cpp` : Includes tests for throwing exceptions and for try/catch constructs with varying number of catch blocks.
- `constructor_tests.cpp` : Contains tests for `constructor`, `destructor`, `new_ptr`, `delete_ptr`, `new_array` and `delete_array`.
- `cast_test.cpp` : Tests for the four cast expressions, as well as `typeid` and `sizeof`.
- `extending_return_type_traits.cpp` : Tests extending the return type deduction system for user defined types. Contains several user defined operators and the corresponding specializations for the return type deduction templates.
- `is_instance_of_test.cpp` : Includes tests for an internally used traits template, which can detect whether a given type is an instance of a certain template or not.
- `bll_and_function.cpp` : Contains tests for using `boost::function` together with lambda functors.

8. Relation to other Boost libraries

8.1. Boost Function

Sometimes it is convenient to store lambda functors in variables. However, the types of even the simplest lambda functors are long and unwieldy, and it is in general unfeasible to declare variables with lambda functor types. *The Boost Function library*[function] defines wrappers for arbitrary function objects, for example lambda functors; and these wrappers have types that are easy to type out. For example:

```
boost::function<int(int, int)> f = _1 + _2;
boost::function<int&(int&)> g = (_1 += 10);
int i = 1, j = 2;
f(i, j); // returns 3
g(i);    // sets i to = 11;
```

The return and parameter types of the wrapped function object must be written explicilty as the template argument to the wrapper template `boost::function`; even when lambda functors, which otherwise have generic parameters, are wrapped. Wrapping a function object with `boost::function` introduces a performance cost comparable to virtual function dispatch, though virtual functions are not actually used. Note that storing lambda functors inside `boost::function` introduces a danger. Certain types of lambda functors may store references to the bound arguments, instead as taking copies of the arguments of the lambda expression. When temporary lambda functor objects are used in STL algorithm invocations this is always safe, as the lambda functor gets destructed immediately after the STL algortihm invocation is completed. However, a lambda functor wrapped inside `boost::function` may continue to exist longer, creating the possibility of dangling references. For example:

```
int* sum = new int();
*sum = 0;
boost::function<int&(int)> counter = *sum += _1;
counter(5); // ok, *sum = 5;
delete sum;
counter(3); // error, *sum does not exist anymore
```

8.2. Boost Bind

The Boost Bind[bind] library has partially overlapping functionality with the BLL. Basically, the Boost Bind library (BB in the sequel) implements the bind expression part of BLL. There are, however, some semantical differerences.

The BLL and BB evolved separately, and have different implementations. This means that the bind expressions from the BB cannot be used within bind expressions, or within other type of lambda expressions, of the BLL. The same holds for using BLL bind expressions in the BB. The libraries can coexist, however, as the names of the BB library are in `boost` namespace, whereas the BLL names are in `boost::lambda` namespace.

The BLL requires a compiler that is reasonably conformant to the C++ standard, whereas the BB library is more portable, and works with a larger set of compilers.

The following two sections describe what are the semantic differences between the bind expressions in BB and BLL.

8.2.1. First argument of bind expression

In BB the first argument of the bind expression, the target function, is treated differently from the other arguments, as no argument substitution takes place within that argument. In BLL the first argument is not a special case in this respect. For example:

```
template<class F>
int foo(const F& f) {
  int x;
  ..
  bind(f, _1)(x);
  ...
}

int bar(int, int);
nested(bind(bar, 1, _1));
```

The bind expression inside `foo` becomes:

```
bind(bind(bar, 1, _1), _1)(x)
```

The BLL interpretes this as:

```
bar(1, x)(x)
```

whereas the BB library as

```
bar(1, x)
```

To get this functionality in BLL, the bind expression inside the `foo` function can be written as:

```
bind(unlambda(f), _1)(x);
```

as explained in Section 5.9.1.1, Unlambda .
The BB library supports up to nine placeholders, while the BLL defines only three placeholders. The rationale for not providing more, is that the highest arity of the function objects accepted by any STL algorithm is two. The placeholder count is easy to increase in the BB library. In BLL it is possible, but more laborous. The BLL currently passes the actual arguments to the lambda functors internally just as they are and does not wrap them inside a tuple object. The reason for this is that some widely used compilers are not capable of optimizing the intermediate tuple objects away. The creation of the intermediate tuples would cause a significant performance hit, particularly for the simplest (and thus the most common) lambda functors. We are working on a hybrid approach, which will allow more placeholders but not compromise the performance of simple lambda functors.

9. Contributors

The main body of the library was written by Jaakko Järvi and Gary Powell. We've got outside help, suggestions and ideas from Jeremy Siek, Peter Higley, Peter Dimov, Valentin Bonnard, William Kempf. We would particularly like to mention Joel de Guzmann and his work with Phoenix which has influenced BLL significantly, making it considerably simpler to extend the library with new features.

Appendix A. Rationale for some of the design decisions

Table of Contents

1. Lambda functor arity

The highest placeholder index in a lambda expression determines the arity of the resulting function object. However, this is just the minimal arity, as the function object can take arbitrarily many arguments; those not needed are discarded. Consider the two bind expressions and their invocations below:

```
bind(g, _3, _3, _3)(x, y, z);
bind(g, _1, _1, _1)(x, y, z);
```

This first line discards arguments `x` and `y`, and makes the call:

```
g(z, z, z)
```

whereas the second line discards arguments `y` and `z`, and calls:

```
g(x, x, x)
```

In earlier versions of the library, the latter line resulted in a compile time error. This is basically a tradeoff between safety and flexibility, and the issue was extensively discussed during the Boost review period of the library. The main points for the *strict arity* checking was that it might catch a programming error at an earlier time and that a lambda expression that explicitly discards its arguments is easy to write:

```
(_3, bind(g, _1, _1, _1))(x, y, z);
```

This lambda expression takes three arguments. The left–hand argument of the comma operator does nothing, and as comma returns the result of evaluating the right–hand argument we end up with the call `g(x, x, x)` even with the strict arity.

The main points against the strict arity checking were that the need to discard arguments is commonplace, and should therefore be straightforward, and that strict arity checking does not really buy that much more safety, particularly as it is not symmetric. For example, if the programmer wanted to write the expression `_1 + _2` but mistakenly wrote `_1 + 2`, with strict arity checking, the complier would spot the error. However, if the erroneous expression was `1 + _2` instead, the error would go unnoticed. Furthermore, weak arity checking simplifies the implementation a bit. Following the recommendation of the Boost review, strict arity checking was dropped.

Bibliography

[STL94] A. A. Stepanov and M. Lee. *The Standard Template Library*. Hewlett–Packard Laboratories. 1994. www.hpl.hp.com/techreports.

[SGI02] *The SGI Standard Template Library*. 2002. www.sgi.com/tech/stl/.

[C++98] *International Standard, Programming Languages C++*. ISO/IEC:14882. 1998.

[Jär99] Jaakko Järvi. *C++ Function Object Binders Made Easy*. . *Lecture Notes in Computer Science*. 1977. Springer. 2000.

[Jär00] Jaakko Järvi. Gary Powell. *The Lambda Library : Lambda Abstraction in C++*. Turku Centre for Computer Science. Technical Report . 378. 2000. www.tucs.fi/publications.

[Jär01] Jaakko Järvi. Gary Powell. *The Lambda Library : Lambda Abstraction in C++*. Second Workshop on C++ Template Programming. Tampa Bay, OOPSLA'01. . 2001. www.oonumerics.org/tmpw01/.

[Jär03] Jaakko Järvi. Gary Powell. Andrew Lumsdaine. *The Lambda Library : unnamed functions in C++*. . *Software − Practice and Expreience*. 33:259−291. 2003.

[tuple] *The Boost Tuple Library*. www.boost.org/libs/tuple/doc/tuple_users_guide.html. 2002.

[type_traits] *The Boost type_traits*. www.boost.org/libs/type_traits/. 2002.

[ref] *Boost ref*. www.boost.org/libs/bind/ref.html. 2002.

[bind] *Boost Bind Library*. www.boost.org/libs/bind/bind.html. 2002.

[function] *Boost Function Library*. www.boost.org/libs/function/. 2002.

[fc++] *The FC++ library: Functional Programming in C++*. Yannis Smaragdakis. Brian McNamara. www.cc.gatech.edu/~yannis/fc++/. 2002.

---

[1] Strictly taken, the C++ standard defines `for_each` as a *non−modifying sequence operation*, and the function object passed to `for_each` should not modify its argument. The requirements for the arguments of `for_each` are unnecessary strict, since as long as the iterators are *mutable*, `for_each` accepts a function object that can have side−effects on their argument. Nevertheless, it is straightforward to provide another function template with the functionality of`std::for_each` but more fine−grained requirements for its arguments.

Chapter 5. Boost.Program_options

Vladimir Prus

1. Introduction

The program_options library allows program developers to obtain *program options*, that is (name, value) pairs from the user, via conventional methods such as command line and config file.

Why would you use such a library, and why is it better than parsing your command line by straightforward hand−written code?

- It's easier. The syntax for declaring options is simple, and the library itself is small. Things like conversion of option values to desired type and storing into program variables are handled automatically.
- Error reporting is better. All the problems with the command line are reported, while hand−written code can just misparse the input. In addition, the usage message can be automatically generated, to avoid falling out of sync with the real list of options.
- Options can be read from anywhere. Sooner or later the command line will be not enough for your users, and you'll want config files or maybe even environment variables. These can be added without significant effort on your part.

Now let's see some examples of the library usage in the Section 2, Tutorial .

2. Tutorial

In this section, we'll take a look at the most common usage scenarios of the program_options library, starting with the simplest one. The examples show only the interesting code parts, but the complete programs can be found in the "BOOST_ROOT/libs/program_options/example" directory. Through all the examples, we'll assume that the following namespace alias is in effect:

```
namespace po = boost::program_options;
```

2.1. Getting Started

The first example is the simplest possible: it only handles two options. Here's the source code (the full program is in "example/first.cpp"):

```
// Declare the supported options.
po::options_description desc("Allowed options");
desc.add_options()
    ("help", "produce help message")
    ("compression", po::value<int>(), "set compression level")
;

po::variables_map vm;
po::store(po::parse_command_line(ac, av, desc), vm);
po::notify(vm);

if (vm.count("help")) {
    cout << desc << "\n";
    return 1;
}
```

```
if (vm.count("compression")) {
    cout << "Compression level was set to "
 << vm["compression"].as<int>() << ".\n";
} else {
    cout << "Compression level was not set.\n";
}
```

We start by declaring all allowed options using the options_description class. The `add_options` method of that class returns a special proxy object that defines `operator()`. Calls to that operator actually declare options. The parameters are option name, information about value, and option description. In this example, the first option has no value, and the second one has a value of type `int`.

After that, an object of class `variables_map` is declared. That class is intended to store values of options, and can store values of arbitrary types. Next, the calls to `store`, `parse_command_line` and `notify` functions cause `vm` to contain all the options found on the command line.

And now, finally, we can use the options as we like. The `variables_map` class can be used just like `std::map`, except that values stored there must be retrieved with the `as` method shown above. (If the type specified in the call to the `as` method is different from the actually stored type, an exception is thrown.)

It's now a good time to try compiling the code yourself, but if you're not yet ready, here's an example session:

```
$bin/gcc/debug/first
Compression level was not set.
$bin/gcc/debug/first --help
Allowed options:
  --help                 : produce help message
  --compression arg      : set compression level

$bin/gcc/debug/first --compression 10
Compression level was set to 10.
```

2.2. Option Details

An option value, surely, can have other types than `int`, and can have other interesting properties, which we'll discuss right now. The complete version of the code snipped below can be found in "example/options_description.cpp".

Imagine we're writing a compiler. It should take the optimization level, a number of include paths, and a number of input files, and perform some interesting work. Let's describe the options:

```
int opt;
po::options_description desc("Allowed options");
desc.add_options()
    ("help", "produce help message")
    ("optimization", po::value<int>(&opt)->default_value(10),
  "optimization level")
    ("include-path,I", po::value< vector<string> >(),
  "include path")
    ("input-file", po::value< vector<string> >(), "input file")
;
```

The "−−help" option should be familiar from the previous example. It's a good idea to have this option in all cases.

The "optimization" option shows two new features. First, we specify the address of the variable(`&opt`). After storing values, that variable will have the value of the option. Second, we specify a default value of 10, which will be used if no value is specified by the user.

The "include−path" option is an example of the only case where the interface of the `options_description` class serves only one source −− the command line. Users typically like to use short option names for common options, and the "include−path,I" name specifies that short option name is "I". So, both "−−include−path" and "−I" can be used.

The "input−file" option specifies the list of files to process. That's okay for a start, but, of course, writing something like:

```
compiler --input-file=a.cpp
```

is a little non−standard, compared with

```
compiler a.cpp
```

We'll address this in a moment.

The command line tokens which have no option name, as above, are called "positional options" by this library. They can be handled too. With a little help from the user, the library can decide that "a.cpp" really means the same as "−−input−file=a.cpp". Here's the additional code we need:

```
po::positional_options_description p;
p.add("input-file", -1);

po::variables_map vm;
po::store(po::command_line_parser(ac, av).
          options(desc).positional(p).run(), vm);
po::notify(vm);
```

The first two lines say that all positional options should be translated into "input−file" options. Also note that we use the command_line_parser class to parse the command line, not the parse_command_line function. The latter is a convenient wrapper for simple cases, but now we need to pass additional information.

By now, all options are described and parsed. We'll save ourselves the trouble of implementing the rest of the compiler logic and only print the options:

```
if (vm.count("include-path"))
{
    cout << "Include paths are: "
         << vm["include-path"].as< vector<string> >() << "\n";
}

if (vm.count("input-file"))
{
    cout << "Input files are: "
         << vm["input-file"].as< vector<string> >() << "\n";
}

cout << "Optimization level is " << opt << "\n";
```

Here's an example session:

```
$bin/gcc/debug/options_description --help
Usage: options_description [options]
Allowed options:
  --help                 : produce help message
  --optimization arg     : optimization level
  -I [ --include-path ] arg : include path
  --input-file arg       : input file
$bin/gcc/debug/options_description
Optimization level is 10
$bin/gcc/debug/options_description --optimization 4 -I foo a.cpp
Include paths are: foo
Input files are: a.cpp
Optimization level is 4
```

Oops, there's a slight problem. It's still possible to specify the "−−input−file" option, and usage message says so, which can be confusing for the user. It would be nice to hide this information, but let's wait for the next example.

2.3. Multiple Sources

It's quite likely that specifying all options to our compiler on the command line will annoy users. What if a user installs a new library and wants to always pass an additional command line element? What if he has made some choices which should be applied on every run? It's desirable to create a config file with common settings which will be used together with the command line.

Of course, there will be a need to combine the values from command line and config file. For example, the optimization level specified on the command line should override the value from the config file. On the other hand, include paths should be combined.

Let's see the code now. The complete program is in "examples/multiple_sources.cpp". The option definition has two interesting details. First, we declare several instances of the `options_description` class. The reason is that, in general, not all options are alike. Some options, like "input−file" above, should not be presented in an automatic help message. Some options make sense only in the config file. Finally, it's nice to have some structure in the help message, not just a long list of options. Let's declare several option groups:

```
// Declare a group of options that will be
// allowed only on command line
po::options_description generic("Generic options");
generic.add_options()
    ("version,v", "print version string")
    ("help", "produce help message")
    ;

// Declare a group of options that will be
// allowed both on command line and in
// config file
po::options_description config("Configuration");
config.add_options()
    ("optimization", po::value<int>(&opt)->default_value(10),
          "optimization level")
    ("include-path,I",
         po::value< vector<string> >()->composing(),
         "include path")
    ;

// Hidden options, will be allowed both on command line and
// in config file, but will not be shown to the user.
po::options_description hidden("Hidden options");
hidden.add_options()
    ("input-file", po::value< vector<string> >(), "input file")
    ;
```

Note the call to the `composing` method in the declaration of the "include−path" option. It tells the library that values from different sources should be composed together, as we'll see shortly.

The `add` method of the `options_description` class can be used to further group the options:

```
po::options_description cmdline_options;
cmdline_options.add(generic).add(config).add(hidden);

po::options_description config_file_options;
config_file_options.add(config).add(hidden);

po::options_description visible("Allowed options");
visible.add(generic).add(config);
```

The parsing and storing of values follows the usual pattern, except that we additionally call parse_config_file, and call the store function twice. But what happens if the same value is specified both on the command line and in config file? Usually, the value stored first is preferred. This is what happens for the "−−optimization" option. For "composing" options, like "include−file", the values are merged.

Here's an example session:

```
$bin/gcc/debug/multiple_sources
Include paths are: /opt
Optimization level is 1
$bin/gcc/debug/multiple_sources --help
Allows options:

Generic options:
  -v [ --version ]       : print version string
  --help                 : produce help message

Configuration:
  --optimization n       : optimization level
  -I [ --include-path ] path : include path

$bin/gcc/debug/multiple_sources --optimization=4 -I foo a.cpp b.cpp
Include paths are: foo /opt
Input files are: a.cpp b.cpp
Optimization level is 4
```

The first invocation uses values from the configuration file. The second invocation also uses values from command line. As we see, the include paths on the command line and in the configuration file are merged, while optimization is taken from the command line.

3. Library Overview

In the tutorial section, we saw several examples of library usage. Here we will describe the overall library design including the primary components and their function.

The library has three main components:

- The options description component, which describes the allowed options and what to do with the values of the options.
- The parsers component, which uses this information to find option names and values in the input sources and return them.
- The storage component, which provides the interface to access the value of an option. It also converts the string representation of values that parsers return into desired C++ types.

To be a little more concrete, the `options_description` class is from the options description component, the `parse_command_line` function is from the parsers component, and the `variables_map` class is from the storage component.

In the tutorial we've learned how those components can be used by the `main` function to parse the command line and config file. Before going into the details of each component, a few notes about the world outside of `main`.

For that outside world, the storage component is the most important. It provides a class which stores all option values and that class can be freely passed around your program to modules which need access to the options. All the other components can be used only in the place where the actual parsing is the done. However, it might also make sense for the individual program modules to describe their options and pass them to the main module, which will merge all options. Of course, this is only important when the number of options is large and declaring them in one place becomes troublesome.

3.1. Options Description Component

The options description component has three main classes: option_description, value_semantic and options_description. The first two together describe a single option. The option_description class contains the option's name, description and a pointer to value_semantic, which, in turn, knows the type of the option's value and can parse the value, apply the default value, and so on. The options_description class is a container for instances of option_description.

For almost every library, those classes could be created in a conventional way: that is, you'd create new options using constructors and then call the `add` method of options_description. However, that's overly verbose for declaring 20 or 30 options. This concern led to creation of the syntax that you've already seen:

```
options_description desc;
desc.add_options()
    ("help", "produce help")
    ("optimization", value<int>()->default_value(10), "optimization level")
    ;
```

The call to the `value` function creates an instance of a class derived from the `value_semantic` class: `typed_value`. That class contains the code to parse values of a specific type, and contains a number of methods which can be called by the user to specify additional information. (This essentially emulates named parameters of the constructor.) Calls to `operator()` on the object returned by `add_options` forward arguments to the constructor of the `option_description` class and add the new instance.

Note that in addition to the `value`, library provides the `bool_switch` function, and user can write his own function which will return other subclasses of `value_semantic` with different behaviour. For the remainder of this section, we'll talk only about the `value` function.

The information about an option is divided into syntactic and semantic. Syntactic information includes the name of the option and the number of tokens which can be used to specify the value. This information is used by parsers to group tokens into (name, value) pairs, where value is just a vector of strings (`std::vector<std::string>`). The semantic layer is responsible for converting the value of the option into more usable C++ types.

This separation is an important part of library design. The parsers use only the syntactic layer, which takes away some of the freedom to use overly complex structures. For example, it's not easy to parse syntax like:

```
calc --expression=1 + 2/3
```

because it's not possible to parse

```
1 + 2/3
```

without knowing that it's a C expression. With a little help from the user the task becomes trivial, and the syntax clear:

```
calc --expression="1 + 2/3"
```

3.1.1. Syntactic information

The syntactic information is provided by the boost::program_options::options_description class and some methods of the boost::program_options::value_semantic class. The simplest usage is illustrated below:

```
          options_description desc;
          desc.add_options()
          ("help", "produce help message")
          ;
```

This declares one option named "help" and associates a description with it. The user is not allowed to specify any value.

To make an option accept a value, you'd need the `value` function mentioned above:

```
options_description desc;
desc.add_options()
("compression", "compression level", value<string>())
("verbose", "verbosity level", value<string>()->implicit())
("email", "email to send to", value<string>()->multitoken());
```

With these declarations, the user must specify a value for the first option, using a single token. For the second option, the user may either provide a single token for the value, or no token at all. For the last option, the value can span several tokens. For example, the following command line is OK:

```
test --compression 10 --verbose --email beadle@mars beadle2@mars
```

3.1.2. Semantic information

The semantic information is completely provided by the boost::program_options::value_semantic class. For example:

```
options_description desc;
desc.add_options()
    ("compression", "compression level", value<int>()->default(10));
    ("email", "email", value< vector<string> >()
        ->composing()->notify(&your_function);
```

These declarations specify that default value of the first option is 10, that the second option can appear several times and all instances should be merged, and that after parsing is done, the library will call function `&your_function`, passing the value of the "email" option as argument.

3.1.3. Positional options

Our definition of option as (name, value) pairs is simple and useful, but in one special case of the command line, there's a problem. A command line can include a *positional option*, which does not specify any name at all, for example:

```
archiver --compression=9 /etc/passwd
```

Here, the "/etc/passwd" element does not have any option name.

One solution is to ask the user to extract positional options himself and process them as he likes. However, there's a nicer approach –– provide a method to automatically assign the names for positional options, so that the above command line can be interpreted the same way as:

```
archiver --compression=9 --input-file=/etc/passwd
```

The positional_options_description class allows the command line parser to assign the names. The class specifies how many positional options are allowed, and for each allowed option, specifies the name. For example:

```
positional_options_description pd; pd.add("input-file", 1, 1);
```

specifies that for exactly one, first, positional option the name will be "input–file".

It's possible to specify that a number, or even all positional options, be given the same name.

```
positional_options_description pd;
pd.add("output-file", 2, 2).add_optional("input-file", 0, -1);
```

In the above example, the first two positional options will be associated with name "output−file", and any others with the name "input−file".

3.2. Parsers Component

The parsers component splits input sources into (name, value) pairs. Each parser looks for possible options and consults the options description component to determine if the option is known and how its value is specified. In the simplest case, the name is explicitly specified, which allows the library to decide if such option is known. If it is known, the value_semantic instance determines how the value is specified. (If it is not known, an exception is thrown.) Common cases are when the value is explicitly specified by the user, and when the value cannot be specified by the user, but the presence of the option implies some value (for example, `true`). So, the parser checks that the value is specified when needed and not specified when not needed, and returns new (name, value) pair.

To invoke a parser you typically call a function, passing the options description and command line or config file or something else. The results of parsing are returned as an instance of the parsed_options class. Typically, that object is passed directly to the storage component. However, it also can be used directly, or undergo some additional processing.

There are three exceptions to the above model −− all related to traditional usage of the command line. While they require some support from the options description component, the additional complexity is tolerable.

- The name specified on the command line may be different from the option name −− it's common to provide a "short option name" alias to a longer name. It's also common to allow an abbreviated name to be specified on the command line.
- Sometimes it's desirable to specify value as several tokens. For example, an option "−−email−recipient" may be followed by several emails, each as a separate command line token. This behaviour is supported, though it can lead to parsing ambiguities and is not enabled by default.
- The command line may contain positional options −− elements which don't have any name. The command line parser provides a mechanism to guess names for such options, as we've seen in the tutorial.

3.3. Storage Component

The storage component is responsible for:

- Storing the final values of an option into a special class and in regular variables
- Handling priorities among different sources.
- Calling user−specified `notify` functions with the final values of options.

Let's consider an example:

```
variables_map vm;
store(parse_command_line(argc, argv, desc), vm);
store(parse_config_file("example.cfg", desc), vm);
notify(vm);
```

The `variables_map` class is used to store the option values. The two calls to the `store` function add values found on the command line and in the config file. Finally the call to the `notify` function runs the user−specified notify functions and stores the values into regular variables, if needed.

The priority is handled in a simple way: the `store` function will not change the value of an option if it's already assigned. In this case, if the command line specifies the value for an option, any value in the config file is ignored.

Warning

Don't forget to call the `notify` function after you've stored all parsed values.

3.4. Annotated List of Symbols

The following table describes all the important symbols in the library, for quick access.

Symbol	Description
Options description component	
options_description	describes a number of options
value	defines the option's value
Parsers component	
parse_command_line	parses command line
parse_config_file	parses config file
parse_environment	parses environment
Storage component	
variables_map	storage for option values

4. How To

This section describes how the library can be used in specific situations.

4.1. Non−conventional Syntax

Sometimes, standard command line syntaxes are not enough. For example, the gcc compiler has "−frtti" and −fno−rtti" options, and this syntax is not directly supported.

For such cases, the library allows the user to provide an *additional parser* −− a function which will be called on each command line element, before any processing by the library. If the additional parser recognises the syntax, it returns the option name and value, which are used directly. The above example can be handled by the following code:

```
pair<string, string> reg_foo(const string& s)
{
    if (s.find("-f") == 0) {
        if (s.substr(2, 3) == "no-")
            return make_pair(s.substr(5), string("false"));
        else
            return make_pair(s.substr(2), string("true"));
    } else {
        return make_pair(string(), string());
    }
}
```

Here's the definition of the additional parser. When parsing the command line, we pass the additional parser:

```
store(command_line_parser(ac, av).options(desc).extra_parser(reg_foo)
        .run(), vm);
```

The complete example can be found in the "example/custom_syntax.cpp" file.

4.2. Response Files

Some operating system have very low limits of the command line length. The common way to work around those limitations is using *response files*. A response file is just a configuration file which uses the same syntax as the command line. If the command line specifies a name of response file to use, it's loaded and parsed in addition to the command line. The library does not provide direct support for response files, so you'll need to write some extra code.

First, you need to define an option for the response file:

```
("response-file", value<string>(),
     "can be specified with '@name', too")
```

Second, you'll need an additional parser to support the standard syntax for specifying response files: "@file":

```
pair<string, string> at_option_parser(string const&s)
{
    if ('@' == s[0])
        return std::make_pair(string("response-file"), s.substr(1));
    else
        return pair<string, string>();
}
```

Finally, when the "response−file" option is found, you'll have to load that file and pass it to the command line parser. This part is the hardest. We'll use the Boost.Tokenizer library, which works but has some limitations. You might also consider Boost.StringAlgo. The code is:

```
if (vm.count("response-file")) {
     // Load the file and tokenize it
     ifstream ifs(vm["response-file"].as<string>().c_str());
     if (!ifs) {
         cout << "Could no open the response file\n";
         return 1;
     }
     // Read the whole file into a string
     stringstream ss;
     ss << ifs.rdbuf();
     // Split the file content
     char_separator<char> sep(" \n\r");
     tokenizer<char_separator<char> > tok(ss.str(), sep);
     vector<string> args;
     copy(tok.begin(), tok.end(), back_inserter(args));
     // Parse the file and store the options
     store(command_line_parser(args).options(desc).run(), vm);
}
```

The complete example can be found in the "example/response_file.cpp" file.

4.3. Winmain Command Line

On the Windows operating system, GUI applications receive the command line as a single string, not split into elements. For that reason, the command line parser cannot be used directly. At least on some compilers, it is possible to obtain the split command line, but it's not clear if all compilers support the same mechanism on all versions of the operating system. The `split_command_line` function is a portable mechanism provided by the library.

Here's an example of use:

```
vector<string> args = split_winmain(lpCmdLine);
store(command_line_parser(args).options(desc).run(), vm);
```

The function is an overload for `wchar_t` strings, so can also be used in Unicode applications.

4.4. Option Groups and Hidden Options

Having a single instance of the options_description class with all the program's options can be problematic:

- Some options make sense only for specific source, for example, configuration files.
- The user would prefer some structure in the generated help message.
- Some options shouldn't appear in the generated help message at all.

To solve the above issues, the library allows a programmer to create several instances of the options_description class, which can be merged in different combinations. The following example will define three groups of options: command line specific, and two options group for specific program modules, only one of which is shown in the generated help message.

Each group is defined using standard syntax. However, you should use reasonable names for each options_description instance:

```
options_description general("General options");
general.add_options()
    ("help", "produce a help message")
    ("help-module", value<string>()->implicit(),
        "produce a help for a given module")
    ("version", "output the version number")
    ;

options_description gui("GUI options");
gui.add_options()
    ("display", value<string>(), "display to use")
    ;

options_description backend("Backend options");
backend.add_options()
    ("num-threads", value<int>(), "the initial number of threads")
    ;
```

After declaring options groups, we merge them in two combinations. The first will include all options and be used for parsing. The second will be used for the "−−help" option.

```
// Declare an options description instance which will include
// all the options
options_description all("Allowed options");
all.add(general).add(gui).add(backend);

// Declare an options description instance which will be shown
// to the user
options_description visible("Allowed options");
visible.add(general).add(gui);
```

What is left is to parse and handle the options:

```
variables_map vm;
store(parse_command_line(ac, av, all), vm);

if (vm.count("help"))
{
    cout << visible;
    return 0;
}
if (vm.count("help-module")) {
    const string& s = vm["help-module"].as<string>();
    if (s == "gui") {
        cout << gui;
    } else if (s == "backend") {
        cout << backend;
    } else {
        cout << "Unknown module '"
             << s << "' in the --help-module option\n";
        return 1;
    }
    return 0;
}
if (vm.count("num-threads")) {
    cout << "The 'num-threads' options was set to "
         << vm["num-threads"].as<int>() << "\n";
}
```

When parsing the command line, all options are allowed. The "−−help" message, however, does not include the "Backend options" group −− the options in that group are hidden. The user can explicitly force the display of that options group by passing "−−help−module backend" option. The complete example can be found in the "example/option_groups.cpp" file.

4.5. Custom Validators

By default, the conversion of option's value from string into C++ type is done using iostreams, which sometimes is not convenient. The library allows the user to customize the conversion for specific classes. In order to do so, the user should provide suitable overload of the `validate` function.

Let's first define a simple class:

```
struct magic_number {
public:
    magic_number(int n) : n(n) {}
    int n;
};
```

and then overload the `validate` function:

```
void validate(boost::any& v,
              const std::vector<std::string>& values,
              magic_number* target_type, int)
{
    static regex r("\\d\\d\\d-(\\d\\d\\d)");

    using namespace boost::program_options;

    // Make sure no previous assignment to 'a' was made.
    validators::check_first_occurence(v);
    // Extract the first string from 'values'. If there is more than
    // one string, it's an error, and exception will be thrown.
    const string& s = validators::get_single_string(values);

    // Do regex match and convert the interesting part to
    // int.
    smatch match;
    if (regex_match(s, match, r)) {
        v = any(magic_number(lexical_cast<int>(match[1])));
    } else {
        throw validation_error("invalid value");
    }
}
```

The function takes four parameters. The first is the storage for the value, and in this case is either empty or contains an instance of the `magic_number` class. The second is the list of strings found in the next occurrence of the option. The remaining two parameters are needed to workaround the lack of partial template specialization and partial function template ordering on some compilers.

The function first checks that we don't try to assign to the same option twice. Then it checks that only a single string was passed in. Next the string is verified with the help of the Boost.Regex library. If that test is passed, the parsed value is stored into the `v` variable.

The complete example can be found in the "example/regex.cpp" file.

4.6. Unicode Support

To use the library with Unicode, you'd need to:

- Use Unicode−aware parsers for Unicode input
- Require Unicode support for options which need it

Most of the parsers have Unicode versions. For example, the parse_command_line function has an overload which takes `wchar_t` strings, instead of ordinary `char`.

Even if some of the parsers are Unicode−aware, it does not mean you need to change definition of all the options. In fact, for many options, like integer ones, it makes no sense. To make use of Unicode you'll need *some* Unicode−aware options. They are different from ordinary options in that they accept `wstring` input, and process it using wide character streams. Creating an Unicode−aware option is easy: just use the the `wvalue` function instead of the regular `value`.

When an ascii parser passes data to an ascii option, or a Unicode parser passes data to a Unicode option, the data are not changed at all. So, the ascii option will see a string in local 8−bit encoding, and the Unicode option will see whatever string was passed as the Unicode input.

What happens when Unicode data is passed to an ascii option, and vice versa? The library automatically performs the conversion from Unicode to local 8−bit encoding. For example, if command line is in ascii, but you use `wstring` options, then the ascii input will be converted into Unicode.

To perform the conversion, the library uses the `codecvt<wchar_t, char>` locale facet from the global locale. If you want to work with strings that use local 8−bit encoding (as opposed to 7−bit ascii subset), your application should start with:

```
locale::global(locale(""));
```

which would set up the conversion facet according to the user's selected locale.

It's wise to check the status of the C++ locale support on your implementation, though. The quick test involves three steps:

1. Go the the "test" directory and build the "test_convert" binary.
2. Set some non−ascii locale in the environmemt. On Linux, one can run, for example:

   ```
   $ export LC_CTYPE=ru_RU.KOI8-R
   ```
3. Run the "test_convert" binary with any non−ascii string in the selected encoding as its parameter. If you see a list of Unicode codepoints, everything's OK. Otherwise, locale support on your system might be broken.

5. Design Discussion

This section focuses on some of the design questions.

5.1. Unicode Support

Unicode support was one of the features specifically requested during the formal review. Throughout this document "Unicode support" is a synonym for "wchar_t" support, assuming that "wchar_t" always uses Unicode encoding. Also, when talking about "ascii" (in lowercase) we'll not mean strict 7−bit ASCII encoding, but rather "char" strings in local 8−bit encoding.

Generally, "Unicode support" can mean many things, but for the program_options library it means that:

- Each parser should accept either `char*` or `wchar_t*`, correctly split the input into option names and option values and return the data.
- For each option, it should be possible to specify whether the conversion from string to value uses ascii or Unicode.
- The library guarantees that:
    - ascii input is passed to an ascii value without change
    - Unicode input is passed to a Unicode value without change
    - ascii input passed to a Unicode value, and Unicode input passed to an ascii value will be converted using a codecvt facet (which may be specified by the user(which can be specified by the user)

The important point is that it's possible to have some "ascii options" together with "Unicode options". There are two reasons for this. First, for a given type you might not have the code to extract the value from Unicode string and it's not good to require that such code be written. Second, imagine a reusable library which has some options and exposes options description in its interface. If *all* options are either ascii or Unicode, and the library does not use any Unicode strings, then the author will likely to use ascii options, which would make the library unusable inside Unicode applications. Essentially, it would be necessary to provide two versions of the library –– ascii and Unicode.

Another important point is that ascii strings are passed though without modification. In other words, it's not possible to just convert ascii to Unicode and process the Unicode further. The problem is that the default conversion mechanism –– the `codecvt` facet –– might not work with 8–bit input without additional setup.

The Unicode support outlined above is not complete. For example, we don't plan allow Unicode in option names. Unicode support is hard and requires a Boost–wide solution. Even comparing two arbitrary Unicode strings is non–trivial. Finally, using Unicode in option names is related to internationalization, which has it's own complexities. E.g. if option names depend on current locale, then all program parts and other parts which use the name must be internationalized too.

The primary question in implementing the Unicode support is whether to use templates and `std::basic_string` or to use some internal encoding and convert between internal and external encodings on the interface boundaries.

The choice, mostly, is between code size and execution speed. A templated solution would either link library code into every application that uses the library (thereby making shared library impossible), or provide explicit instantiations in the shared library (increasing its size). The solution based on internal encoding would necessarily make conversions in a number of places and will be somewhat slower. Since speed is generally not an issue for this library, the second solution looks more attractive, but we'll take a closer look at individual components.

For the parsers component, we have three choices:

- Use a fully templated implementation: given a string of a certain type, a parser will return a parsed_options instance with strings of the same type (i.e. the parsed_options class will be templated).
- Use internal encoding: same as above, but strings will be converted to and from the internal encoding.
- Use and partly expose the internal encoding: same as above, but the strings in the parsed_options instance will be in the internal encoding. This might avoid a conversion if parsed_options instance is passed directly to other components, but can be also dangerous or confusing for a user.

The second solution appears to be the best –– it does not increase the code size much and is cleaner than the third. To avoid extra conversions, the Unicode version of parsed_options can also store strings in internal encoding.

For the options descriptions component, we don't have much choice. Since it's not desirable to have either all options use ascii or all of them use Unicode, but rather have some ascii and some Unicode options, the interface of the value_semantic must work with both. The only way is to pass an additional flag telling if strings use ascii or internal encoding. The instance of value_semantic can then convert into some other encoding if needed.

For the storage component, the only affected function is store. For Unicode input, the store function should convert the value to the internal encoding. It should also inform the value_semantic class about the used encoding.

Finally, what internal encoding should we use? The alternatives are: `std::wstring` (using UCS−4 encoding) and `std::string` (using UTF−8 encoding). The difference between alternatives is:

- Speed: UTF−8 is a bit slower
- Space: UTF−8 takes less space when input is ascii
- Code size: UTF−8 requires additional conversion code. However, it allows one to use existing parsers without converting them to `std::wstring` and such conversion is likely to create a number of new instantiations.

There's no clear leader, but the last point seems important, so UTF−8 will be used.

Choosing the UTF−8 encoding allows the use of existing parsers, because 7−bit ascii characters retain their values in UTF−8, so searching for 7−bit strings is simple. However, there are two subtle issues:

- We need to assume the character literals use ascii encoding and that inputs use Unicode encoding.
- A Unicode character (say '=') can be followed by 'composing character' and the combination is not the same as just '=', so a simple search for '=' might find the wrong character.

Neither of these issues appear to be critical in practice, since ascii is almost universal encoding and since composing characters following '=' (and other characters with special meaning to the library) are not likely to appear.

6. Acknowledgements

I'm very gratefull to all the people who helped with the development, by discussion, fixes, and as users. It was pleasant to see all that involvement, which made the library much better than it would be otherwise.

In the early stages, the library was affected by discussions with Gennadiy Rozental, William Kempf and Alexander Okhotin.

Hartmut Kaiser was the first person to try the library on his project and send a number of suggestions and fixes.

The formal review lead to numerous comments and enhancements. Pavol Droba helped with the option description semantic. Gennadiy Rozental has criticised many aspects of the library which caused various simplifications. Pavel Vozenilek did carefull review of the implementation. A number of comments were made by:

- David Abrahams
- Neal D. Becker
- Misha Bergal
- James Curran
- Carl Daniel
- Beman Dawes
- Tanton Gibbs
- Holger Grund
- Hartmut Kaiser
- Petr Kocmid
- Baptiste Lepilleur
- Marcelo E. Magallon
- Chuck Messenger
- John Torjo
- Matthias Troyer

Doug Gregor and Reece Dunn helped to resolve the issues with Boostbook version of the documentation.

Even after review, a number of people have helped with further development:

- Rob Lievaart
- Thorsten Ottosen
- Joseph Wu

- Ferdinand Prantl
- Miro Jurisic
- John Maddock
- Janusz Piwowarski
- Charles Brockman
- Jonathan Wakely

7. Reference

7.1. Header <boost/program_options/cmdline.hpp>

```
namespace boost {
  namespace program_options {
    namespace command_line_style {
      enum style_t;
    }
  }
}
```

Type style_t

boost::program_options::command_line_style::style_t

Synopsis

```
enum style_t { allow_long =  1, allow_short =  allow_long << 1, allow_dash_for_short =  allow_short << 1,
               allow_slash_for_short =  allow_dash_for_short << 1, long_allow_adjacent =  allow_slash_for_sho
               short_allow_adjacent =  long_allow_next << 1, short_allow_next =  short_allow_adjacent << 1, a
               allow_guessing =  allow_sticky << 1, case_insensitive =  allow_guessing << 1, allow_long_disgu
               unix_style =  (allow_short | short_allow_adjacent | short_allow_next
                      | allow_long | long_allow_adjacent | long_allow_next
                      | allow_sticky | allow_guessing
                      | allow_dash_for_short), default_style =  unix_style };
```

7.2. Header <boost/program_options/environment_iterator.hpp>

```
namespace boost {
  class environment_iterator;
}
```

Class environment_iterator

boost::environment_iterator

Synopsis

```
class environment_iterator : public boost::eof_iterator< environment_iterator, std::pair< std::string, std::s
{
public:
  // construct/copy/destruct
  environment_iterator(char **);
  environment_iterator();

  // public member functions
  void get() ;
};
```

Description

`environment_iterator` construct/copy/destruct

1. `environment_iterator(char ** environment);`
2. `environment_iterator();`

`environment_iterator` public member functions

1. `void get() ;`

7.3. Header <boost/program_options/eof_iterator.hpp>

```
namespace boost {
  template<typename Derived, typename ValueType> class eof_iterator;
}
```

Class template eof_iterator

boost::eof_iterator

Synopsis

```
template<typename Derived, typename ValueType>
class eof_iterator {
public:
  // construct/copy/destruct
  eof_iterator();

  // public member functions

  // protected member functions
  ValueType & value() ;
  void found_eof() ;

  // private member functions
  void increment() ;
  bool equal(const eof_iterator &) const;
  const ValueType & dereference() const;
};
```

Description

The 'eof_iterator' class is useful for constructing forward iterators in cases where iterator extract data from some source and it's easy to detect 'eof' –– i.e. the situation where there's no data. One apparent example is reading lines from a file.

Implementing such iterators using 'iterator_facade' directly would require to create class with three core operation, a couple of constructors. When using 'eof_iterator', the derived class should define only one method to get new value, plus a couple of constructors.

The basic idea is that iterator has 'eof' bit. Two iterators are equal only if both have their 'eof' bits set. The 'get' method either obtains the new value or sets the 'eof' bit.

Specifically, derived class should define:

1. A default constructor, which creates iterator with 'eof' bit set. The constructor body should call 'found_eof' method defined here. 2. Some other constructor. It should initialize some 'data pointer' used in iterator operation and then call 'get'. 3. The

'get' method. It should operate this way:

- look at some 'data pointer' to see if new element is available; if not, it should call 'found_eof'.
- extract new element and store it at location returned by the 'value' method.
- advance the data pointer.

Essentially, the 'get' method has the functionality of both 'increment' and 'dereference'. It's very good for the cases where data extraction implicitly moves data pointer, like for stream operation.

`eof_iterator` construct/copy/destruct

1. `eof_iterator();`

`eof_iterator` public member functions

`eof_iterator` protected member functions

1. `ValueType & value() ;`

   Returns the reference which should be used by derived class to store the next value.
2. `void found_eof() ;`

   Should be called by derived class to indicate that it can't produce next element.

`eof_iterator` private member functions

1. `void increment() ;`
2. `bool equal(const eof_iterator & other) const;`
3. `const ValueType & dereference() const;`

7.4. Header <boost/program_options/errors.hpp>

```
namespace boost {
  namespace program_options {
    class error;
    class invalid_syntax;
    class unknown_option;
    class ambiguous_option;
    class multiple_values;
    class multiple_occurrences;
    class validation_error;
    class invalid_option_value;
    class too_many_positional_options_error;
    class too_few_positional_options_error;
    class invalid_command_line_syntax;
    class invalid_command_line_style;
  }
}
```

Class error

boost::program_options::error

Synopsis

```
class error {
public:
  // construct/copy/destruct
  error(const std::string &);

  // public member functions
};
```

Description

Base class for all errors in the library.

`error` construct/copy/destruct

1. `error(const std::string & what);`

`error` public member functions

Class invalid_syntax

boost::program_options::invalid_syntax

Synopsis

```
class invalid_syntax : public boost::program_options::error {
public:
  // construct/copy/destruct
  invalid_syntax(const std::string &, const std::string &);
  ~invalid_syntax();

  // public member functions

  std::string tokens;
  std::string msg;
};
```

Description

`invalid_syntax` construct/copy/destruct

1. `invalid_syntax(const std::string & tokens, const std::string & msg);`
2. `~invalid_syntax();`

`invalid_syntax` public member functions

Class unknown_option

boost::program_options::unknown_option

Synopsis

```
class unknown_option : public boost::program_options::error {
public:
  // construct/copy/destruct
  unknown_option(const std::string &);

  // public member functions
};
```

Description

Class thrown when option name is not recognized.

`unknown_option` construct/copy/destruct

1. `unknown_option(const std::string & name);`

`unknown_option` public member functions

Class ambiguous_option

boost::program_options::ambiguous_option

Synopsis

```
class ambiguous_option : public boost::program_options::error {
public:
  // construct/copy/destruct
  ambiguous_option(const std::string &, const std::vector< std::string > &);
  ~ambiguous_option();

  // public member functions

  std::vector< std::string > alternatives;
};
```

Description

Class thrown when there's ambiguity amoung several possible options.

`ambiguous_option` construct/copy/destruct

1.
   ```
   ambiguous_option(const std::string & name,
                    const std::vector< std::string > & alternatives);
   ```
2. `~ambiguous_option();`

`ambiguous_option` public member functions

Class multiple_values

boost::program_options::multiple_values

Synopsis

```
class multiple_values : public boost::program_options::error {
public:
  // construct/copy/destruct
  multiple_values(const std::string &);

  // public member functions
};
```

Description

Class thrown when there are several option values, but user called a method which cannot return them all.

`multiple_values` construct/copy/destruct

1. `multiple_values(const std::string & what);`

`multiple_values` public member functions

Class multiple_occurrences

boost::program_options::multiple_occurrences

Synopsis

```
class multiple_occurrences : public boost::program_options::error {
public:
  // construct/copy/destruct
  multiple_occurrences(const std::string &);

  // public member functions
};
```

Description

Class thrown when there are several occurrences of an option, but user called a method which cannot return them all.

`multiple_occurrences` construct/copy/destruct

1. `multiple_occurrences(const std::string & what);`

`multiple_occurrences` public member functions

Class validation_error

boost::program_options::validation_error

Synopsis

```
class validation_error : public boost::program_options::error {
public:
  // construct/copy/destruct
  validation_error(const std::string &);
  ~validation_error();

  // public member functions
  void set_option_name(const std::string &) ;

  // private member functions
  const char * what() const;
};
```

Description

Class thrown when value of option is incorrect.

`validation_error` construct/copy/destruct

1. `validation_error(const std::string & what);`
2. `~validation_error();`

`validation_error` public member functions

1. `void set_option_name(const std::string & option) ;`

`validation_error` private member functions

1. `const char * what() const;`

Class invalid_option_value

boost::program_options::invalid_option_value

Synopsis

```
class invalid_option_value
  :  : public boost::program_options::validation_error
{
public:
  // construct/copy/destruct
  invalid_option_value(const std::string &);
  invalid_option_value(const std::wstring &);

  // public member functions
};
```

Description

`invalid_option_value` construct/copy/destruct

1. `invalid_option_value(const std::string & value);`
2. `invalid_option_value(const std::wstring & value);`

`invalid_option_value` public member functions

Class too_many_positional_options_error

boost::program_options::too_many_positional_options_error

Synopsis

```
class too_many_positional_options_error
  :  : public boost::program_options::error
{
public:
  // construct/copy/destruct
  too_many_positional_options_error(const std::string &);

  // public member functions
};
```

Description

Class thrown when there are too many positional options.

`too_many_positional_options_error` construct/copy/destruct

1. `too_many_positional_options_error(const std::string & what);`

`too_many_positional_options_error` public member functions

Class too_few_positional_options_error

boost::program_options::too_few_positional_options_error

Synopsis

```
class too_few_positional_options_error
  :  : public boost::program_options::error
{
public:
  // construct/copy/destruct
  too_few_positional_options_error(const std::string &);

  // public member functions
};
```

Description

Class thrown when there are too few positional options.

`too_few_positional_options_error` construct/copy/destruct

1. `too_few_positional_options_error(const std::string & what);`

`too_few_positional_options_error` public member functions

Class invalid_command_line_syntax

boost::program_options::invalid_command_line_syntax

Synopsis

```
class invalid_command_line_syntax {
public:
  // construct/copy/destruct
  invalid_command_line_syntax(const std::string &, kind_t);

  // public member functions
  kind_t kind() const;

  // protected static functions
  std::string error_message(kind_t) ;
};
```

Description

`invalid_command_line_syntax` construct/copy/destruct

1. `invalid_command_line_syntax(const std::string & tokens, kind_t kind);`

`invalid_command_line_syntax` public member functions

1. `kind_t kind() const;`

`invalid_command_line_syntax` protected static functions

1. `std::string error_message(kind_t kind) ;`

Class invalid_command_line_style

boost::program_options::invalid_command_line_style

Synopsis

```
class invalid_command_line_style : public boost::program_options::error {
public:
  // construct/copy/destruct
  invalid_command_line_style(const std::string &);

  // public member functions
};
```

Description

`invalid_command_line_style` construct/copy/destruct

1. `invalid_command_line_style(const std::string & msg);`

`invalid_command_line_style` public member functions

7.5. Header <boost/program_options/option.hpp>

```
namespace boost {
  namespace program_options {
    template<typename charT> class basic_option;

    typedef basic_option< char > option;
    typedef basic_option< wchar_t > woption;
  }
}
```

Class template basic_option

boost::program_options::basic_option

Synopsis

```
template<typename charT>
class basic_option {
public:
  // construct/copy/destruct
  basic_option();
  basic_option(const std::string &, const std::vector< std::string > &);

  // public member functions

  std::string string_key;
  int position_key;
  std::vector< std::basic_string< charT > > value;
};
```

Description

Option found in input source. Contains a key and a value. The key, in turn, can be a string (name of an option), or an integer (position in input source) –– in case no name is specified. The latter is only possible for command line. The template parameter specifies the type of char used for storing the option's value.

`basic_option` construct/copy/destruct

1. `basic_option();`

2.
   ```
   basic_option(const std::string & string_key,
                const std::vector< std::string > & value);
   ```

`basic_option` public member functions

7.6. Header <boost/program_options/options_description.hpp>

```
namespace boost {
  namespace program_options {
    class option_description;
    class options_description_easy_init;
```

```
    class options_description;
    class duplicate_option_error;
  }
}
```

Class option_description

boost::program_options::option_description

Synopsis

```
class option_description {
public:
  // construct/copy/destruct
  option_description();
  option_description(const char *, const value_semantic *);
  option_description(const char *, const value_semantic *, const char *);
  ~option_description();

  // public member functions
  const std::string & short_name() const;
  const std::string & long_name() const;
  const std::string & description() const;
  shared_ptr< const value_semantic > semantic() const;
  std::string format_name() const;
  std::string format_parameter() const;

  // private member functions
  option_description & name(const char *) ;
};
```

Description

Describes one possible command line/config file option. There are two kinds of properties of an option. First describe it syntactically and are used only to validate input. Second affect interpretation of the option, for example default value for it or function that should be called when the value is finally known. Routines which perform parsing never use second kind of properties –– they are side effect free.

options_description

`option_description` construct/copy/destruct

1. `option_description();`
2. `option_description(const char * name, const value_semantic * s);`

   Initializes the object with the passed data.

   Note: it would be nice to make the second parameter auto_ptr, to explicitly pass ownership. Unfortunately, it's often needed to create objects of types derived from 'value_semantic': options_description d; d.add_options()("a", parameter<int>("n")–>default_value(1)); Here, the static type returned by 'parameter' should be derived from value_semantic.

   Alas, derived–>base conversion for auto_ptr does not really work, see
   http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2000/n1232.pdf
   http://std.dkuug.dk/jtc1/sc22/wg21/docs/cwg_defects.html#84

   So, we have to use plain old pointers. Besides, users are not expected to use the constructor directly.

The 'name' parameter is interpreted by the following rules:

- if there's no "," character in 'name', it specifies long name
- otherwise, the part before "," specifies long name and the part after –– long name.

3.
```
option_description(const char * name, const value_semantic * s,
                   const char * description);
```

   Initializes the class with the passed data.
4. `~option_description();`

`option_description` public member functions

1. `const std::string & short_name() const;`
2. `const std::string & long_name() const;`
3. `const std::string & description() const;`
4. `shared_ptr< const value_semantic > semantic() const;`
5. `std::string format_name() const;`
6. `std::string format_parameter() const;`

   Return the parameter name and properties, formatted suitably for usage message.

`option_description` private member functions

1. `option_description & name(const char * name) ;`

Class options_description_easy_init

boost::program_options::options_description_easy_init

Synopsis

```
class options_description_easy_init {
public:
  // construct/copy/destruct
  options_description_easy_init(options_description *);

  // public member functions
  options_description_easy_init & operator()(const char *, const char *) ;
  options_description_easy_init &
  operator()(const char *, const value_semantic *) ;
  options_description_easy_init &
  operator()(const char *, const value_semantic *, const char *) ;
};
```

Description

Class which provides convenient creation syntax to option_description.

`options_description_easy_init` construct/copy/destruct

1. `options_description_easy_init(options_description * owner);`

`options_description_easy_init` public member functions

1.
   ```
   options_description_easy_init &
   operator()(const char * name, const char * description) ;
   ```

2.
   ```
   options_description_easy_init &
   operator()(const char * name, const value_semantic * s) ;
   ```

3.
   ```
   options_description_easy_init &
   operator()(const char * name, const value_semantic * s,
              const char * description) ;
   ```

Class options_description

boost::program_options::options_description

Synopsis

```
class options_description {
public:
  // construct/copy/destruct
  options_description();
  options_description(const std::string &);

  // public member functions
  void add(shared_ptr< option_description >) ;
  options_description & add(const options_description &) ;
  options_description_easy_init add_options() ;
  unsigned count(const std::string &) const;
  unsigned count_approx(const std::string &) const;
  const option_description & find(const std::string &) const;
  const option_description & find_approx(const std::string &) const;
  std::set< std::string > keys() const;
  std::set< std::string > primary_keys() const;
  std::set< std::string > approximations(const std::string &) const;
  void print(std::ostream &) const;

  // private member functions
  approximation_range find_approximation(const std::string &) const;
};
```

Description

A set of option descriptions. This provides convenient interface for adding new option (the add_options) method, and facilities to search for options by name.

See here for option adding interface discussion.

option_description

`options_description` construct/copy/destruct

1. `options_description();`

   Creates the instance.
2. `options_description(const std::string & caption);`

   Creates the instance. The 'caption' parameter gives the name of this 'options_description' instance. Primarily useful for output.

`options_description` public member functions

1. `void add(shared_ptr< option_description > desc) ;`

   Adds new variable description. Throws duplicate_variable_error if either short or long name matches that of already present one.
2. `options_description & add(const options_description & desc) ;`

   Adds a group of option description. This has the same effect as adding all option_descriptions in 'desc' individually, except that output operator will show a separate group. Returns *this.
3. `options_description_easy_init add_options() ;`

   Returns an object of implementation−defined type suitable for adding options to options_description. The returned object will have overloaded operator() with parameter type matching 'option_description' constructors. Calling the operator will create new option_description instance and add it.
4. `unsigned count(const std::string & name) const;`

   Count the number of option descriptions with given name. Returns 0 or 1. The 'name' parameter can be either name of long option, and short option prefixed by '−'.
5. `unsigned count_approx(const std::string & prefix) const;`

   Count the number of descriptions having the given string as prefix. This makes sense only for long options.
6. `const option_description & find(const std::string & name) const;`

   Returns description given a name.

   Requires

   count(name) == 1
7. `const option_description & find_approx(const std::string & prefix) const;`

   Returns description given a prefix. Throws

   Requires

   count_approx(name) == 1
8. `std::set< std::string > keys() const;`
9. `std::set< std::string > primary_keys() const;`

   For each option description stored, contains long name if not empty, if it is empty, short name is returned.
10. `std::set< std::string > approximations(const std::string & prefix) const;`
11. `void print(std::ostream & os) const;`

    Output 'desc' to the specified stream, calling 'f' to output each option_description element.

`options_description` private member functions

1. `approximation_range find_approximation(const std::string & prefix) const;`

Class duplicate_option_error

boost::program_options::duplicate_option_error

Synopsis

```
class duplicate_option_error : public boost::program_options::error {
public:
  // construct/copy/destruct
  duplicate_option_error(const std::string &);
  // public member functions
};
```

Description

Class thrown when duplicate option description is found.

`duplicate_option_error` construct/copy/destruct

1. `duplicate_option_error(const std::string & what);`

`duplicate_option_error` public member functions

7.7. Header <boost/program_options/parsers.hpp>

```
namespace boost {
  namespace program_options {
    template<typename charT> class basic_parsed_options;

    template<> class basic_parsed_options<wchar_t>;

    class common_command_line_parser;
    template<typename charT> class basic_command_line_parser;

    typedef basic_parsed_options< char > parsed_options;
    typedef basic_parsed_options< wchar_t > wparsed_options;
    typedef function1< std::pair< std::string, std::string >, const std::string & > ext_parser;
    typedef basic_command_line_parser< char > command_line_parser;
    typedef basic_command_line_parser< wchar_t > wcommand_line_parser;
    template<typename charT>
      basic_parsed_options< charT >
      parse_command_line(int, charT *, const options_description &, int = 0,
                         function1< std::pair< std::string, std::string >, const std::string & > = ext_parse
    template<typename charT>
      BOOST_PROGRAM_OPTIONS_DECL basic_parsed_options< charT >
      parse_config_file(std::basic_istream< charT > &,
                        const options_description &);
    BOOST_PROGRAM_OPTIONS_DECL parsed_options
    parse_environment(const options_description &,
                      const function1< std::string, std::string > &);
    BOOST_PROGRAM_OPTIONS_DECL parsed_options
    parse_environment(const options_description &, const std::string &);
    BOOST_PROGRAM_OPTIONS_DECL parsed_options
    parse_environment(const options_description &, const char *);
  }
}
```

Class template basic_parsed_options

boost::program_options::basic_parsed_options

Synopsis

```
template<typename charT>
class basic_parsed_options {
public:
  // construct/copy/destruct
  basic_parsed_options(const options_description *);

  // public member functions
  std::vector< basic_option< charT > > options;
  const options_description * description;
};
```

Description

Results of parsing an input source. The primary use of this class is passing information from parsers component to value storage component. This class does not makes much sense itself.

`basic_parsed_options` construct/copy/destruct

1. `basic_parsed_options(const options_description * description);`

`basic_parsed_options` public member functions

Specializations

- Class basic_parsed_options<wchar_t>

Class basic_parsed_options<wchar_t>

boost::program_options::basic_parsed_options<wchar_t>

Synopsis

```
class basic_parsed_options<wchar_t> {
public:

  // public member functions
   basic_parsed_options(const basic_parsed_options< char > &) ;

  std::vector< basic_option< wchar_t > > options;
  const options_description * description;
  basic_parsed_options< char > utf8_encoded_options;
};
```

Description

Specialization of basic_parsed_options which:

- provides convenient conversion from basic_parsed_options<char>
- stores the passed char−based options for later use.

`basic_parsed_options` public member functions

1. `basic_parsed_options(const basic_parsed_options< char > & po) ;`

   Constructs wrapped options from options in UTF8 encoding.

Class common_command_line_parser

boost::program_options::common_command_line_parser

Synopsis

```
class common_command_line_parser {
public:
  // construct/copy/destruct
  common_command_line_parser(const std::vector< std::string > &);

  // public member functions
  parsed_options run() const;
};
```

Description

Character−type independent command line parser.

`common_command_line_parser` construct/copy/destruct

1. `common_command_line_parser(const std::vector< std::string > & args);`

`common_command_line_parser` public member functions

1. `parsed_options run() const;`

   Parses the command line and returns parsed options in internal encoding.

Class template basic_command_line_parser

boost::program_options::basic_command_line_parser

Synopsis

```
template<typename charT>
class basic_command_line_parser
  :  : private boost::program_options::common_command_line_parser
{
public:
  // construct/copy/destruct
  basic_command_line_parser(const std::vector< std::basic_string< charT > > &);
  basic_command_line_parser(int, charT *);

  // public member functions
  basic_command_line_parser & options(const options_description &) ;
  basic_command_line_parser &
  positional(const positional_options_description &) ;
  basic_command_line_parser & style(int) ;
  basic_command_line_parser & extra_parser(ext_parser) ;
  basic_parsed_options< charT > run() const;
};
```

Description

Command line parser.

The class allows one to specify all the information needed for parsing and to parser the parse the command line. It is primarily needed to emulate named function parameters -- a regular function with 5 parameters will be hard to use and creating overloads with a smaller nuber of parameters will be confusing.

For the most common case, the function parse_command_line is a better alternative.

`basic_command_line_parser` construct/copy/destruct

1. `basic_command_line_parser(const std::vector< std::basic_string< charT > > & args);`

   Creates a command line parser for the specified arguments list. The 'args' parameter should not include program name.
2. `basic_command_line_parser(int argc, charT * argv);`

   Creates a command line parser for the specified arguments list. The parameter should be the same as passes to 'main'.

`basic_command_line_parser` public member functions

1. `basic_command_line_parser & options(const options_description & desc) ;`

   Sets options descriptions to use.

2.
   ```
   basic_command_line_parser &
   positional(const positional_options_description & desc) ;
   ```

   Sets positional options description to use.
3. `basic_command_line_parser & style(int ) ;`

   Sets the command line style.
4. `basic_command_line_parser & extra_parser(ext_parser ) ;`

   Sets the extra parsers.
5. `basic_parsed_options< charT > run() const;`

   Parses the command line and returns parsed options in internal encoding.

Function template parse_command_line

boost::program_options::parse_command_line

Synopsis

```
template<typename charT>
  basic_parsed_options< charT >
  parse_command_line(int argc, charT * argv, const options_description & ,
                     int style = 0,
                     function1< std::pair< std::string, std::string >, const std::string & > ext = ext_parse
```

Description

Creates instance of 'command_line_parser', passes parameters to it, and returns the result of calling the 'run' method.

Function template parse_config_file

boost::program_options::parse_config_file

Synopsis

```
template<typename charT>
  BOOST_PROGRAM_OPTIONS_DECL basic_parsed_options< charT >
  parse_config_file(std::basic_istream< charT > & ,
                    const options_description & );
```

Description

Parse a config file.

Function parse_environment

boost::program_options::parse_environment

Synopsis

```
BOOST_PROGRAM_OPTIONS_DECL parsed_options
parse_environment(const options_description & ,
                  const function1< std::string, std::string > & name_mapper);
```

Description

Parse environment.

For each environment variable, the 'name_mapper' function is called to obtain the option name. If it returns empty string, the variable is ignored.

This is done since naming of environment variables is typically different from the naming of command line options.

Function parse_environment

boost::program_options::parse_environment

Synopsis

```
BOOST_PROGRAM_OPTIONS_DECL parsed_options
parse_environment(const options_description & , const std::string & prefix);
BOOST_PROGRAM_OPTIONS_DECL parsed_options
parse_environment(const options_description & , const char * prefix);
```

Description

Parse environment.

Takes all environment variables which start with 'prefix'. The option name is obtained from variable name by removing the prefix and converting the remaining string into lower case.

7.8. Header <boost/program_options/positional_options.hpp>

```
namespace boost {
  namespace program_options {
    class positional_options_description;
  }
}
```

Class positional_options_description

boost::program_options::positional_options_description

Synopsis

```
class positional_options_description {
public:
  // construct/copy/destruct
  positional_options_description();

  // public member functions
  void add(const char *, int) ;
  unsigned max_total_count() const;
  const std::string & name_for_position(unsigned) const;
};
```

Description

Describes positional options.

The class allows to guess option names for positional options, which are specified on the command line and are identified by the position. The class uses the information provided by the user to associate a name with every positional option, or tell that no name is known.

The primary assumption is that only the relative order of the positional options themselves matters, and that any interleaving ordinary options don't affect interpretation of positional options.

The user initializes the class by specifying that first N positional options should be given the name X1, following M options should be given the name X2 and so on.

`positional_options_description` construct/copy/destruct

1. `positional_options_description();`

`positional_options_description` public member functions

1. `void add(const char * name, int max_count) ;`

   Species that up to 'max_count' next positional options should be given the 'name'. The value of '−1' means 'unlimited'. No calls to 'add' can be made after call with 'max_value' equal to '−1'.
2. `unsigned max_total_count() const;`

   Returns the maximum number of positional options that can be present. Can return numeric_limits<unsigned>::max() to indicate unlimited number.
3. `const std::string & name_for_position(unsigned position) const;`

Returns the name that should be associated with positional options at 'position'. Precondition: position < max_total_count()

7.9. Header <boost/program_options/value_semantic.hpp>

```
namespace boost {
  namespace program_options {
    class value_semantic;
    template<typename charT> class value_semantic_codecvt_helper;

    template<> class value_semantic_codecvt_helper<char>;
    template<> class value_semantic_codecvt_helper<wchar_t>;

    class untyped_value;
    template<typename T, typename charT = char> class typed_value;
    template<typename T> typed_value< T > * value();
    template<typename T> typed_value< T > * value(T *);
    template<typename T> typed_value< T, wchar_t > * wvalue();
    template<typename T> typed_value< T, wchar_t > * wvalue(T *);
    BOOST_PROGRAM_OPTIONS_DECL typed_value< bool > * bool_switch();
    BOOST_PROGRAM_OPTIONS_DECL typed_value< bool > * bool_switch(bool *);
  }
}
```

Class value_semantic

boost::program_options::value_semantic

Synopsis

```
class value_semantic {
public:
  // construct/copy/destruct
  ~value_semantic();

  // public member functions
  virtual std::string name() const;
  virtual bool is_zero_tokens() const;
  virtual bool is_composing() const;
  virtual bool is_implicit() const;
  virtual bool is_multitoken() const;
  virtual void
  parse(boost::any &, const std::vector< std::string > &, bool) const;
  virtual bool apply_default(boost::any &) const;
  virtual void notify(const boost::any &) const;
};
```

Description

Class which specifies how the option's value is to be parsed and converted into C++ types.

`value_semantic` construct/copy/destruct

1. `~value_semantic();`

`value_semantic` public member functions

1. `virtual std::string name() const;`

Returns the name of the option. The name is only meaningful for automatic help message.

2. `virtual bool is_zero_tokens() const;`

   Returns true if value cannot be specified in source at all. Other methods can still set the value somehow, but user can't affect it.
3. `virtual bool is_composing() const;`

   Returns true if values from different sources should be composed. Otherwise, value from the first source is used and values from other sources are discarded.
4. `virtual bool is_implicit() const;`

   Returns true if explicit value of an option can be omitted.
5. `virtual bool is_multitoken() const;`

   Returns true if value can span several token in input source.

6.
   ```
   virtual void
   parse(boost::any & value_store, const std::vector< std::string > & new_tokens,
         bool utf8) const;
   ```

   Parses a group of tokens that specify a value of option. Stores the result in 'value_store', using whatever representation is desired. May be be called several times if value of the same option is specified more than once.
7. `virtual bool apply_default(boost::any & value_store) const;`

   Called to assign default value to 'value_store'. Returns true if default value is assigned, and false if no default value exists.
8. `virtual void notify(const boost::any & value_store) const;`

   Called when final value of an option is determined.

Class template value_semantic_codecvt_helper

boost::program_options::value_semantic_codecvt_helper

Synopsis

```
template<typename charT>
class value_semantic_codecvt_helper {
public:
};
```

Description

Helper class which perform necessary character conversions in the 'parse' method and forwards the data further.

Specializations

- Class value_semantic_codecvt_helper<char>
- Class value_semantic_codecvt_helper<wchar_t>

Class value_semantic_codecvt_helper<char>

boost::program_options::value_semantic_codecvt_helper<char>

Synopsis

```
class value_semantic_codecvt_helper<char> {
public:

  // protected member functions
  virtual void xparse(boost::any &, const std::vector< std::string > &) const;

  // private member functions
  void parse(boost::any &, const std::vector< std::string > &, bool) const;
};
```

Description

`value_semantic_codecvt_helper` protected member functions

1.
   ```
   virtual void
   xparse(boost::any & value_store,
          const std::vector< std::string > & new_tokens) const;
   ```

`value_semantic_codecvt_helper` private member functions

1.
   ```
   void parse(boost::any & value_store,
              const std::vector< std::string > & new_tokens, bool utf8) const;
   ```

Class value_semantic_codecvt_helper<wchar_t>

boost::program_options::value_semantic_codecvt_helper<wchar_t>

Synopsis

```
class value_semantic_codecvt_helper<wchar_t>
  :  : public boost::program_options::value_semantic
{
public:

  // protected member functions
  virtual void xparse(boost::any &, const std::vector< std::wstring > &) const;

  // private member functions
  void parse(boost::any &, const std::vector< std::string > &, bool) const;
};
```

Description

`value_semantic_codecvt_helper` protected member functions

1.
   ```
   virtual void
   xparse(boost::any & value_store,
          const std::vector< std::wstring > & new_tokens) const;
   ```

`value_semantic_codecvt_helper` private member functions

1.
   ```
   void parse(boost::any & value_store,
              const std::vector< std::string > & new_tokens, bool utf8) const;
   ```

Class untyped_value

boost::program_options::untyped_value

Synopsis

```
class untyped_value
  :  : public boost::program_options::value_semantic_codecvt_helper< charT >
{
public:
  // construct/copy/destruct
  untyped_value(bool = false);

  // public member functions
  std::string name() const;
  bool is_zero_tokens() const;
  bool is_composing() const;
  bool is_implicit() const;
  bool is_multitoken() const;
  void xparse(boost::any &, const std::vector< std::string > &) const;
  bool apply_default(boost::any &) const;
  void notify(const boost::any &) const;
};
```

Description

Class which specifies a simple handling of a value: the value will have string type and only one token is allowed.

`untyped_value` construct/copy/destruct

1. `untyped_value(bool zero_tokens = false);`

`untyped_value` public member functions

1. `std::string name() const;`
2. `bool is_zero_tokens() const;`
3. `bool is_composing() const;`
4. `bool is_implicit() const;`
5. `bool is_multitoken() const;`

6.
   ```
   void xparse(boost::any & value_store,
               const std::vector< std::string > & new_tokens) const;
   ```

   If 'value_store' is already initialized, or new_tokens has more than one elements, throws. Otherwise, assigns the first string from 'new_tokens' to 'value_store', without any modifications.
7. `bool apply_default(boost::any & ) const;`

   Does nothing.
8. `void notify(const boost::any & ) const;`

   Does nothing.

Class template typed_value

boost::program_options::typed_value

Synopsis

```
template<typename T, typename charT = char>
class typed_value
  :  : public boost::program_options::value_semantic_codecvt_helper< charT >
{
public:
  // construct/copy/destruct
  typed_value(T *);

  // public member functions
  typed_value * default_value(const T &) ;
  typed_value * default_value(const T &, const std::string &) ;
  typed_value * notifier(function1< void, const T & >) ;
  typed_value * composing() ;
  typed_value * implicit() ;
  typed_value * multitoken() ;
  typed_value * zero_tokens() ;
  std::string name() const;
  bool is_zero_tokens() const;
  bool is_composing() const;
  bool is_implicit() const;
  bool is_multitoken() const;
  void xparse(boost::any &, const std::vector< std::basic_string< charT > > &) const;
  virtual bool apply_default(boost::any &) const;
  void notify(const boost::any &) const;
};
```

Description

Class which handles value of a specific type.

`typed_value` construct/copy/destruct

1. `typed_value(T * store_to);`

   Ctor. The 'store_to' parameter tells where to store the value when it's known. The parameter can be NULL.

`typed_value` public member functions

1. `typed_value * default_value(const T & v) ;`

   Specifies default value, which will be used if none is explicitly specified. The type 'T' should provide operator<< for ostream.
2. `typed_value * default_value(const T & v, const std::string & textual) ;`

   Specifies default value, which will be used if none is explicitly specified. Unlike the above overload, the type 'T' need not provide operator<< for ostream, but textual representation of default value must be provided by the user.
3. `typed_value * notifier(function1< void, const T & > f) ;`

   Specifies a function to be called when the final value is determined.
4. `typed_value * composing() ;`

   Specifies that the value is composing. See the 'is_composing' method for explanation.
5. `typed_value * implicit() ;`

   Specifies that the value is implicit.
6. `typed_value * multitoken() ;`

   Specifies that the value can span multiple tokens.

7. `typed_value * zero_tokens() ;`
8. `std::string name() const;`
9. `bool is_zero_tokens() const;`
10. `bool is_composing() const;`
11. `bool is_implicit() const;`
12. `bool is_multitoken() const;`
13.
```
void xparse(boost::any & value_store, 
            const std::vector< std::basic_string< charT > > & new_tokens) const;
```

    Creates an instance of the 'validator' class and calls its operator() to perform athe ctual conversion.
14. `virtual bool apply_default(boost::any & value_store) const;`

    If default value was specified via previous call to 'default_value', stores that value into 'value_store'. Returns true if default value was stored.
15. `void notify(const boost::any & value_store) const;`

    If an address of variable to store value was specified when creating *this, stores the value there. Otherwise, does nothing.

Function value

boost::program_options::value

Synopsis

```
template<typename T> typed_value< T > * value();
template<typename T> typed_value< T > * value(T * v);
```

Description

Creates a typed_value<T> instance. This function is the primary method to create value_semantic instance for a specific type, which can later be passed to 'option_description' constructor. The second overload is used when it's additionally desired to store the value of option into program variable.

Function wvalue

boost::program_options::wvalue

Synopsis

```
template<typename T> typed_value< T, wchar_t > * wvalue();
template<typename T> typed_value< T, wchar_t > * wvalue(T * v);
```

Description

Creates a typed_value<T> instance. This function is the primary method to create value_semantic instance for a specific type, which can later be passed to 'option_description' constructor.

Function bool_switch

boost::program_options::bool_switch

Synopsis

```
BOOST_PROGRAM_OPTIONS_DECL typed_value< bool > * bool_switch();
BOOST_PROGRAM_OPTIONS_DECL typed_value< bool > * bool_switch(bool * v);
```

Description

Works the same way as the 'value<bool>' function, but the created value_semantic won't accept any explicit value. So, if the option is present on the command line, the value will be 'true'.

7.10. Header <boost/program_options/variables_map.hpp>

```
namespace boost {
  namespace program_options {
    class variable_value;
    class abstract_variables_map;
    class variables_map;
    BOOST_PROGRAM_OPTIONS_DECL void
    store(const basic_parsed_options< char > &, variables_map &, bool = false);
    BOOST_PROGRAM_OPTIONS_DECL void
    store(const basic_parsed_options< wchar_t > &, variables_map &);
    BOOST_PROGRAM_OPTIONS_DECL void notify(variables_map &);
  }
}
```

Class variable_value

boost::program_options::variable_value

Synopsis

```
class variable_value {
public:
  // construct/copy/destruct
  variable_value();
  variable_value(const boost::any &, bool);

  // public member functions
  template<typename T> const T & as() const;
  template<typename T> T & as() ;
  bool empty() const;
  bool defaulted() const;
  const boost::any & value() const;
  boost::any & value() ;
};
```

Description

Class holding value of option. Contains details about how the value is set and allows to conveniently obtain the value.

`variable_value` construct/copy/destruct

1. `variable_value();`
2. `variable_value(const boost::any & v, bool defaulted);`

`variable_value` public member functions

1. `template<typename T> const T & as() const;`

   If stored value if of type T, returns that value. Otherwise, throws boost::bad_any_cast exception.
2. `template<typename T> T & as() ;`

   This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
3. `bool empty() const;`
4. `bool defaulted() const;`

   Returns true if the value was not explicitly given, but has default value.
5. `const boost::any & value() const;`

   Returns the contained value.
6. `boost::any & value() ;`

   Returns the contained value.

Class abstract_variables_map

boost::program_options::abstract_variables_map

Synopsis

```
class abstract_variables_map {
public:
  // construct/copy/destruct
  abstract_variables_map();
  abstract_variables_map(const abstract_variables_map *);
  ~abstract_variables_map();

  // public member functions
  const variable_value & operator[](const std::string &) const;
  void next(abstract_variables_map *) ;

  // private member functions
  virtual const variable_value & get(const std::string &) const;
};
```

Description

Implements string−>string mapping with convenient value casting facilities.

`abstract_variables_map` construct/copy/destruct

1. `abstract_variables_map();`
2. `abstract_variables_map(const abstract_variables_map * next);`
3. `~abstract_variables_map();`

`abstract_variables_map` public member functions

1. `const variable_value & operator[](const std::string & name) const;`

   Obtains the value of variable 'name', from *this and possibly from the chain of variable maps.

- if there's no value in *this.
  - if there's next variable map, returns value from it
  - otherwise, returns empty value
- if there's defaulted value
  - if there's next varaible map, which has a non–defauled value, return that
  - otherwise, return value from *this
- if there's a non–defauled value, returns it.

2. `void next(abstract_variables_map * next) ;`

   Sets next variable map, which will be used to find variables not found in *this.

`abstract_variables_map` private member functions

1. `virtual const variable_value & get(const std::string & name) const;`

   Returns value of variable 'name' stored in *this, or empty value otherwise.

Class variables_map

boost::program_options::variables_map

Synopsis

```
class variables_map
  :  : public boost::program_options::abstract_variables_map
{
public:
  // construct/copy/destruct
  variables_map();
  variables_map(const abstract_variables_map *);

  // public member functions
  const variable_value & operator[](const std::string &) const;

  // private member functions
  const variable_value & get(const std::string &) const;
};
```

Description

Concrete variables map which store variables in real map.

`variables_map` construct/copy/destruct

1. `variables_map();`
2. `variables_map(const abstract_variables_map * next);`

`variables_map` public member functions

1. `const variable_value & operator[](const std::string & name) const;`

   Obtains the value of variable 'name', from *this and possibly from the chain of variable maps.

- if there's no value in *this.
    - if there's next variable map, returns value from it
    - otherwise, returns empty value
- if there's defaulted value
    - if there's next varaible map, which has a non−defauled value, return that
    - otherwise, return value from *this
- if there's a non−defauled value, returns it.

`variables_map` private member functions

1. `const variable_value & get(const std::string & name) const;`

   Implementation of abstract_variables_map::get which does 'find' in *this.

Function store

boost::program_options::store

Synopsis

```
BOOST_PROGRAM_OPTIONS_DECL void
store(const basic_parsed_options< char > & options, variables_map & m,
      bool utf8 = false);
```

Description

Stores in 'm' all options that are defined in 'options'. If 'm' already has a non−defaulted value of an option, that value is not changed, even if 'options' specify some value.

Function store

boost::program_options::store

Synopsis

```
BOOST_PROGRAM_OPTIONS_DECL void
store(const basic_parsed_options< wchar_t > & options, variables_map & m);
```

Description

Stores in 'm' all options that are defined in 'options'. If 'm' already has a non−defaulted value of an option, that value is not changed, even if 'options' specify some value. This is wide character variant.

Function notify

boost::program_options::notify

Synopsis

```
BOOST_PROGRAM_OPTIONS_DECL void notify(variables_map & m);
```

Description

Runs all 'notify' function for options in 'm'.

7.11. Header <boost/program_options/version.hpp>

```
BOOST_PROGRAM_OPTIONS_VERSION
```

Macro BOOST_PROGRAM_OPTIONS_VERSION

BOOST_PROGRAM_OPTIONS_VERSION

Synopsis

```
BOOST_PROGRAM_OPTIONS_VERSION
```

Description

The version of the source interface. The value will be incremented whenever a change is made which might cause compilation errors for existing code.

Chapter 6. Boost.Ref

Jaakko Järvi

Peter Dimov

Douglas Gregor

Dave Abrahams

1. 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>`, the two functions `boost::ref` and `boost::cref` that return instances of `boost::reference_wrapper<T>`, 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::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.

2. Reference

2.1. Header <boost/ref.hpp>

```
namespace boost {
  template<typename T> class reference_wrapper;
  reference_wrapper<T> ref(T&);
  reference_wrapper<T const> cref(T const&);
  template<typename T> class is_reference_wrapper;
  template<typename T> class unwrap_reference;
}
```

Class template reference_wrapper

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

Synopsis

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

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

  // access
  operator T&() const;
  T& get() const;
  T* get_pointer() const;
};

// constructors
reference_wrapper<T> ref(T&);
reference_wrapper<T const> cref(T 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` construct/copy/destruct

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

   Effects
      Constructs a `reference_wrapper` object that stores a reference to `t`.
   Throws
      Does not throw.

`reference_wrapper` access

1. `operator T&() const;`

   Returns
      The stored reference.
   Throws
      Does not throw.
2. `T& get() const;`

   Returns
      The stored reference.
   Throws
      Does not throw.
3. `T* get_pointer() const;`

   Returns
      A pointer to the object referenced by the stored reference.

Throws
: Does not throw.

`reference_wrapper` constructors

1. `reference_wrapper<T> ref(T& t);`

   Returns
   : `reference_wrapper<T>(t)`

   Throws
   : Does not throw.

2. `reference_wrapper<T const> cref(T const& t);`

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

   Throws
   : Does not throw.

Class template is_reference_wrapper

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

Synopsis

```
template<typename T>
class is_reference_wrapper {
public:
  // static constants
  static const bool value = unspecified;
};
```

Description

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

Class template unwrap_reference

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

Synopsis

```
template<typename T>
class unwrap_reference {
public:
  // types
  typedef unspecified type;
};
```

Description

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

3. Acknowledgements

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.

Chapter 7. Boost.Signals

Douglas Gregor

1. Introduction

The Boost.Signals library is an implementation of a managed signals and slots system. Signals represent callbacks with multiple targets, and are also called publishers or events in similar systems. Signals are connected to some set of slots, which are callback receivers (also called event targets or subscribers), which are called when the signal is "emitted."

Signals and slots are managed, in that signals and slots (or, more properly, objects that occur as part of the slots) track all connections and are capable of automatically disconnecting signal/slot connections when either is destroyed. This enables the user to make signal/slot connections without expending a great effort to manage the lifetimes of those connections with regard to the lifetimes of all objects involved.

When signals are connected to multiple slots, there is a question regarding the relationship between the return values of the slots and the return value of the signals. Boost.Signals allows the user to specify the manner in which multiple return values are combined.

2. Tutorial

2.1. How to Read this Tutorial

This tutorial is not meant to be read linearly. Its top−level structure roughly separates different concepts in the library (e.g., handling calling multiple slots, passing values to and from slots) and in each of these concepts the basic ideas are presented first and then more complex uses of the library are described later. Each of the sections is marked *Beginner*, *Intermediate*, or *Advanced* to help guide the reader. The *Beginner* sections include information that all library users should know; one can make good use of the Signals library after having read only the *Beginner* sections. The *Intermediate* sections build on the *Beginner* sections with slightly more complex uses of the library. Finally, the *Advanced* sections detail very advanced uses of the Signals library, that often require a solid working knowledge of the *Beginner* and *Intermediate* topics; most users will not need to read the *Advanced* sections.

2.2. Compatibility Note

Boost.Signals has two syntactical forms: the preferred form and the compatibility form. The preferred form fits more closely with the C++ language and reduces the number of separate template parameters that need to be considered, often improving readability; however, the preferred form is not supported on all platforms due to compiler bugs. The compatible form will work on all compilers supported by Boost.Signals. Consult the table below to determine which syntactic form to use for your compiler. Users of Boost.Function, please note that the preferred syntactic form in Signals is equivalent to that of Function's preferred syntactic form.

If your compiler does not appear in this list, please try the preferred syntax and report your results to the Boost list so that we can keep this table up−to−date.

Preferred syntax	**Portable syntax**
• GNU C++ 2.95.x, 3.0.x, 3.1.x	• *Any compiler supporting the preferred syntax*

• Comeau C++ 4.2.45.2 • SGI MIPSpro 7.3.0 • Intel C++ 5.0, 6.0 • Compaq's cxx 6.2 • Microsoft Visual C++ 7.1	• Microsoft Visual C++ 6.0, 7.0 • Borland C++ 5.5.1 • Sun WorkShop 6 update 2 C++ 5.3 • Metrowerks CodeWarrior 8.1

2.3. Hello, World! (Beginner)

The following example writes "Hello, World!" using signals and slots. First, we create a signal `sig`, a signal that takes no arguments and has a void return value. Next, we connect the `hello` function object to the signal using the `connect` method. Finally, use the signal `sig` like a function to call the slots, which in turns invokes `HelloWorld::operator()` to print "Hello, World!".

Preferred syntax	Portable syntax

```
struct HelloWorld
{
  void operator()() const
  {
    std::cout << "Hello, World!" << std::endl;
  }
};

// ...

// Signal with no arguments and a void return value
boost::signal<void ()> sig;

// Connect a HelloWorld slot
HelloWorld hello;
sig.connect(hello);

// Call all of the slots
sig();
```
	```
	struct HelloWorld
	{
	  void operator()() const
	  {
	    std::cout << "Hello, World!" << std::endl;
	  }
	};

	// ...

	// Signal with no arguments and a void return value
	boost::signal0<void> sig;

	// Connect a HelloWorld slot
	HelloWorld hello;
	sig.connect(hello);

	// Call all of the slots
	sig();
	```

2.4. Calling multiple slots

2.4.1. Connecting multiple slots (Beginner)

Calling a single slot from a signal isn't very interesting, so we can make the Hello, World program more interesting by splitting the work of printing "Hello, World!" into two completely separate slots. The first slot will print "Hello" and may look like this:

```
struct Hello
{
  void operator()() const
  {
    std::cout << "Hello";
  }
};
```

The second slot will print ", World!" and a newline, to complete the program. The second slot may look like this:

```
struct World
{
  void operator()() const
  {
    std::cout << ", World!" << std::endl;
  }
};
```

Like in our previous example, we can create a signal `sig` that takes no arguments and has a `void` return value. This time, we connect both a `hello` and a `world` slot to the same signal, and when we call the signal both slots will be called.

Preferred syntax	Portable syntax

`boost::signal<void ()> sig;`
 	
`sig.connect(Hello());`
`sig.connect(World());`
 	
`sig();`
	`boost::signal0<void> sig;`
	
	`sig.connect(Hello());`
	`sig.connect(World());`
	
	`sig();`

By default, slots are called in first–in first–out (FIFO) order, so the output of this program will be as expected:

```
Hello, World!
```

2.4.2. Ordering slot call groups (Intermediate)

Slots are free to have side effects, and that can mean that some slots will have to be called before others even if they are not connected in that order. The Boost.Signals library allows slots to be placed into groups that are ordered in some way. For our Hello, World program, we want "Hello" to be printed before ", World!", so we put "Hello" into a group that must be executed before the group that ", World!" is in. To do this, we can supply an extra parameter at the beginning of the `connect` call that specifies the group. Group values are, by default, `int`s, and are ordered by the integer < relation. Here's how we construct Hello, World:

Preferred syntax	Portable syntax

`boost::signal<void ()> sig;`
`sig.connect(1, World());`
`sig.connect(0, Hello());`
`sig();`
	`boost::signal0<void> sig;`
	`sig.connect(1, World());`
	`sig.connect(0, Hello());`
	`sig();`

This program will correctly print "Hello, World!", because the `Hello` object is in group 0, which precedes group 1 where the `World` object resides. The group parameter is, in fact, optional. We omitted it in the first Hello, World example because it was unnecessary when all of the slots are independent. So what happens if we mix calls to connect that use the group parameter and those that don't? The "unnamed" slots (i.e., those that have been connected without specifying a group name) can be placed at the front or back of the slot list (by passing `boost::signals::at_front` or `boost::signals::at_back` as the last parameter to `connect`, respectively), and defaults to the end of the list. When a group is specified, the final parameter describes where the slot will be placed within the group ordering. If we add a new slot to our example like this:

```
struct GoodMorning
{
  void operator()() const
  {
    std::cout << "... and good morning!" << std::endl;
  }
};

sig.connect(GoodMorning());
```

... we will get the result we wanted:

```
Hello, World!
... and good morning!
```

2.5. Passing values to and from slots

2.5.1. Slot Arguments (Beginner)

Signals can propagate arguments to each of the slots they call. For instance, a signal that propagates mouse motion events might want to pass along the new mouse coordinates and whether the mouse buttons are pressed.

As an example, we'll create a signal that passes two `float` arguments to its slots. Then we'll create a few slots that print the results of various arithmetic operations on these values.

```
void print_sum(float x, float y)
{
  std::cout << "The sum is " << x+y << std::endl;
}

void print_product(float x, float y)
{
  std::cout << "The product is " << x*y << std::endl;
}

void print_difference(float x, float y)
{
  std::cout << "The difference is " << x-y << std::endl;
}

void print_quotient(float x, float y)
{
  std::cout << "The quotient is " << x/y << std::endl;
}
```

Preferred syntax	Portable syntax

`boost::signal<void (float, float)> sig;`
 	
`sig.connect(&print_sum);`
`sig.connect(&print_product);`
`sig.connect(&print_difference);`
`sig.connect(&print_quotient);`
 	
`sig(5, 3);`
	`boost::signal2<void, float, float> sig;`
	
	`sig.connect(&print_sum);`
	`sig.connect(&print_product);`
	`sig.connect(&print_difference);`
	`sig.connect(&print_quotient);`
	
	`sig(5, 3);`

This program will print out the following:

```
The sum is 8
The difference is 2
The product is 15
The quotient is 1.66667
```

So any values that are given to `sig` when it is called like a function are passed to each of the slots. We have to declare the types of these values up front when we create the signal. The type `boost::signal<void (float, float)>` means that the signal has a `void` return value and takes two `float` values. Any slot connected to `sig` must therefore be able to take two `float` values.

2.5.2. Signal Return Values (Advanced)

Just as slots can receive arguments, they can also return values. These values can then be returned back to the caller of the signal through a *combiner*. The combiner is a mechanism that can take the results of calling slots (there many be no results or a hundred; we don't know until the program runs) and coalesces them into a single result to be returned to the caller. The single result is often a simple function of the results of the slot calls: the result of the last slot call, the maximum value returned by any slot, or a container of all of the results are some possibilities.

We can modify our previous arithmetic operations example slightly so that the slots all return the results of computing the product, quotient, sum, or difference. Then the signal itself can return a value based on these results to be printed:

Preferred syntax	Portable syntax

`float product(float x, float y) { return x*y; }`
`float quotient(float x, float y) { return x/y; }`
`float sum(float x, float y) { return x+y; }`
`float difference(float x, float y) { return x-y; }`
 	
`boost::signal<float (float x, float y)> sig;`
 	
`sig.connect(&product);`
`sig.connect(&quotient);`
`sig.connect(&sum);`
`sig.connect(&difference);`
 	
`std::cout << sig(5, 3) << std::endl;`
	`float product(float x, float y) { return x*y; }`
	`float quotient(float x, float y) { return x/y; }`
	`float sum(float x, float y) { return x+y; }`
	`float difference(float x, float y) { return x-y; }`
	
	`boost::signal2<float, float, float> sig;`
	
	`sig.connect(&product);`
	`sig.connect(&quotient);`
	`sig.connect(&sum);`
	`sig.connect(&difference);`
	
	`std::cout << sig(5, 3) << std::endl;`

This example program will output `2`. This is because the default behavior of a signal that has a return type (`float`, the first template argument given to the `boost::signal` class template) is to call all slots and then return the result returned by the last slot called. This behavior is admittedly silly for this example, because slots have no side effects and the result is the last slot connect.

A more interesting signal result would be the maximum of the values returned by any slot. To do this, we create a custom combiner that looks like this:

```
template<typename T>
struct maximum
{
  typedef T result_type;

  template<typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    // If there are no slots to call, just return the
    // default-constructed value
    if (first == last)
      return T();

    T max_value = *first++;
    while (first != last) {
      if (max_value < *first)
        max_value = *first;
      ++first;
    }

    return max_value;
  }
};
```

The `maximum` class template acts as a function object. Its result type is given by its template parameter, and this is the type it expects to be computing the maximum based on (e.g., `maximum<float>` would find the maximum `float` in a sequence of `float`s). When a `maximum` object is invoked, it is given an input iterator sequence `[first, last)` that includes the results of calling all of the slots. `maximum` uses this input iterator sequence to calculate the maximum element, and returns that maximum value.

We actually use this new function object type by installing it as a combiner for our signal. The combiner template argument follows the signal's calling signature:

Preferred syntax	Portable syntax

`boost::signal<float (float x, float y), maximum<float> > sig;`	`boost::signal2<float, float, float, maximum<float> > sig;`

Now we can connect slots that perform arithmetic functions and use the signal:

```
sig.connect(&quotient);
sig.connect(&product);
sig.connect(&sum);
sig.connect(&difference);

std::cout << sig(5, 3) << std::endl;
```

The output of this program will be `15`, because regardless of the order in which the slots are connected, the product of 5 and 3 will be larger than the quotient, sum, or difference.

In other cases we might want to return all of the values computed by the slots together, in one large data structure. This is easily done with a different combiner:

```
template<typename Container>
struct aggregate_values
{
  typedef Container result_type;

  template<typename InputIterator>
  Container operator()(InputIterator first, InputIterator last) const
  {
    return Container(first, last);
  }
};
```

Again, we can create a signal with this new combiner:

Preferred syntax	**Portable syntax**

`boost::signal<float (float, float), aggregate_values<std::vector<float> > > sig;`
 	
`sig.connect(&quotient);`
`sig.connect(&product);`
`sig.connect(&sum);`
`sig.connect(&difference);`
 	
`std::vector<float> results = sig(5, 3);`
`std::copy(results.begin(), results.end(), std::ostream_iterator<float>(cout, " "));`
	`boost::signal2<float, float, float, aggregate_values<std::vector<float> > > sig;`
	
	`sig.connect(&quotient);`
	`sig.connect(&product);`
	`sig.connect(&sum);`
	`sig.connect(&difference);`
	
	`std::vector<float> results = sig(5, 3);`
	`std::copy(results.begin(), results.end(), std::ostream_iterator<float>(cout, " "));`

The output of this program will contain 15, 8, 1.6667, and 2. It is interesting here that the first template argument for the `signal` class, `float`, is not actually the return type of the signal. Instead, it is the return type used by the connected slots and will also be the `value_type` of the input iterators passed to the combiner. The combiner itself is a function object and its `result_type` member type becomes the return type of the signal.

The input iterators passed to the combiner transform dereference operations into slot calls. Combiners therefore have the option to invoke only some slots until some particular criterion is met. For instance, in a distributed computing system, the combiner may ask each remote system whether it will handle the request. Only one remote system needs to handle a particular request, so after a remote system accepts the work we do not want to ask any other remote systems to perform the same task. Such a combiner need only check the value returned when dereferencing the iterator, and return when the value is acceptable. The following combiner returns the first non–NULL pointer to a `FulfilledRequest` data structure, without asking any later slots to fulfill the request:

```
struct DistributeRequest {
  typedef FulfilledRequest* result_type;
  template<typename InputIterator>
  result_type operator()(InputIterator first, InputIterator last) const
  {
    while (first != last) {
      if (result_type fulfilled = *first)
        return fulfilled;
      ++first;
    }
    return 0;
  }
};
```

2.6. Connection Management

2.6.1. Disconnecting Slots (Beginner)

Slots aren't expected to exist indefinately after they are connected. Often slots are only used to receive a few events and are then disconnected, and the programmer needs control to decide when a slot should no longer be connected.

The entry point for managing connections explicitly is the `boost::signals::connection` class. The `connection` class uniquely represents the connection between a particular signal and a particular slot. The `connected()` method checks if the signal and slot are still connected, and the `disconnect()` method disconnects the signal and slot if they are connected before it is called. Each call to the signal's `connect()` method returns a connection object, which can be used to determine if the connection still exists or to disconnect the signal and slot.

```
boost::signals::connection c = sig.connect(HelloWorld());
if (c.connected()) {
// c is still connected to the signal
  sig(); // Prints "Hello, World!"
}

c.disconnect(); // Disconnect the HelloWorld object
assert(!c.connected()); c isn't connected any more

sig(); // Does nothing: there are no connected slots
```

2.6.2. Scoped connections (Intermediate)

The `boost::signals::scoped_connection` class references a signal/slot connection that will be disconnected when the `scoped_connection` class goes out of scope. This ability is useful when a connection need only be temporary, e.g.,

```
{
  boost::signals::scoped_connection c = sig.connect(ShortLived());
  sig(); // will call ShortLived function object
}
sig(); // ShortLived function object no longer connected to sig
```

2.6.3. Disconnecting equivalent slots (Intermediate)

One can disconnect slots that are equivalent to a given function object using a form of the `disconnect` method, so long as the type of the function object has an accessible == operator. For instance:

Preferred syntax	**Portable syntax**
`void foo();` `void bar();` `signal<void()> sig;`	`void foo();` `void bar();` `signal0<void> sig;`

`sig.connect(&foo);`	`sig.connect(&foo);`
`sig.connect(&bar);`	`sig.connect(&bar);`
`// disconnects foo, but not bar`	`// disconnects foo, but not bar`
`sig.disconnect(&foo);`	`sig.disconnect(&foo);`

2.6.4. Automatic connection management (Intermediate)

Boost.Signals can automatically track the lifetime of objects involved in signal/slot connections, including automatic disconnection of slots when objects involved in the slot call are destroyed. For instance, consider a simple news delivery service, where clients connect to a news provider that then sends news to all connected clients as information arrives. The news delivery service may be constructed like this:

Preferred syntax	**Portable syntax**
`class NewsItem { /* ... */ };`	`class NewsItem { /* ... */ };`
`boost::signal<void (const NewsItem&)> deliverNews;`	`boost::signal1<void, const NewsItem&> deliverNews;`

Clients that wish to receive news updates need only connect a function object that can receive news items to the `deliverNews` signal. For instance, we may have a special message area in our application specifically for news, e.g.,:

```
struct NewsMessageArea : public MessageArea
{
public:
  // ...

  void displayNews(const NewsItem& news) const
  {
    messageText = news.text();
    update();
  }
};

// ...
NewsMessageArea newsMessageArea = new NewsMessageArea(/* ... */);
// ...
deliverNews.connect(boost::bind(&NewsMessageArea::displayNews,
                                newsMessageArea, _1));
```

However, what if the user closes the news message area, destroying the `newsMessageArea` object that `deliverNews` knows about? Most likely, a segmentation fault will occur. However, with Boost.Signals one need only make `NewsMessageArea`*trackable*, and the slot involving `newsMessageArea` will be disconnected when `newsMessageArea` is destroyed. The `NewsMessageArea` class is made trackable by deriving publicly from the `boost::signals::trackable` class, e.g.:

```
struct NewsMessageArea : public MessageArea, public boost::signals::trackable
{
  // ...
};
```

At this time there is a significant limitation to the use of `trackable` objects in making slot connections: function objects built using Boost.Bind are understood, such that pointers or references to `trackable` objects passed to `boost::bind` will be found and tracked.

Warning: User–defined function objects and function objects from other libraries (e.g., Boost.Function or Boost.Lambda) do not implement the required interfaces for `trackable` object detection, and *will silently ignore any bound trackable objects*. Future versions of the Boost libraries will address this limitation.

2.6.5. When can disconnections occur? (Intermediate)

Signal/slot disconnections occur when any of these conditions occur:

- The connection is explicitly disconnected via the connection's `disconnect` method directly, or indirectly via the signal's `disconnect` method or `scoped_connection`'s destructor.
- A `trackable` object bound to the slot is destroyed.
- The signal is destroyed.

These events can occur at any time without disrupting a signal's calling sequence. If a signal/slot connection is disconnected at any time during a signal's calling sequence, the calling sequence will still continue but will not invoke the disconnected slot. Additionally, a signal may be destroyed while it is in a calling sequence, and which case it will complete its slot call sequence but may not be accessed directly.

Signals may be invoked recursively (e.g., a signal A calls a slot B that invokes signal A...). The disconnection behavior does not change in the recursive case, except that the slot calling sequence includes slot calls for all nested invocations of the signal.

2.6.6. Passing slots (Intermediate)

Slots in the Boost.Signals library are created from arbitrary function objects, and therefore have no fixed type. However, it is commonplace to require that slots be passed through interfaces that cannot be templates. Slots can be passed via the `slot_type` for each particular signal type and any function object compatible with the signature of the signal can be passed to a `slot_type` parameter. For instance:

Preferred syntax

```
class Button
{
  typedef boost::signal<void (int x, int y)> OnClick;

public:
  void doOnClick(const OnClick::slot_type& slot);

private:
  OnClick onClick;
};

void Button::doOnClick(
      const OnClick::slot_type& slot
    )
{
  onClick.connect(slot);
}

void printCoordinates(long x, long y)
{
  std::cout << "(" << x << ", " << y << ")\n";
}

void f(Button& button)
{
  button.doOnClick(&printCoordinates);
}
```

Portable syntax

```
class Button
{
  typedef boost::signal2<void,int,int> OnClick;

public:
  void doOnClick(const OnClick::slot_type& slot);

private:
  OnClick onClick;
};

void Button::doOnClick(
      const OnClick::slot_type& slot
    )
{
  onClick.connect(slot);
}

void printCoordinates(long x, long y)
{
  std::cout << "(" << x << ", " << y << ")\n";
}

void f(Button& button)
{
  button.doOnClick(&printCoordinates);
}
```

The `doOnClick` method is now functionally equivalent to the `connect` method of the `onClick` signal, but the details of the `doOnClick` method can be hidden in an implementation detail file.

2.7. Linking against the Signals library

Part of the Boost.Signals library is compiled into a binary library that must be linked into your application to use Signals. To build this library, execute the command **bjam** in either the top–level Boost directory or in `libs/signals/build`. On Unix, the directory `libs/signals/build/bin-stage` will then contain libraries named, e.g., `libboost_signals.a` that can be linked in your program with `-lboost_signals`.

On Windows, with Microsoft Visual C++ or Borland C++, the linking process is nearly automatic. As with the Regex library, the libraries in `libs\signals\build\bin-stage` will have mangled names and will be automatically be including in the link process. To link against the Signals library binary dynamically (e.g., using the Signals DLL), define `BOOST_SIGNALS_DYN_LINK` when building your application; to link statically, define `BOOST_SIGNALS_STATIC_LINK`.

3. Reference

3.1. Header <boost/signal.hpp>

```
namespace boost {
  template<typename R, typename T1, typename T2, ..., typename TN,
           typename Combiner = last_value<R>, typename Group = int,
           typename GroupCompare = std::less<Group>,
           typename SlotFunction = functionN<R, T1, T2, ..., TN> >
    class signalN;
  template<typename Signature, typename Combiner = last_value<R>,
           typename Group = int, typename GroupCompare = std::less<Group>,
           typename SlotFunction = functionN<Signature> >
    class signal;
  namespace signals {

    enum connect_position { at_front, at_back };
  }
}
```

Class template signalN

boost::signalN   Set of safe multicast callback types.

Synopsis

```
template<typename R, typename T1, typename T2, ..., typename TN,
         typename Combiner = last_value<R>, typename Group = int,
         typename GroupCompare = std::less<Group>,
         typename SlotFunction = functionN<R, T1, T2, ..., TN> >
class signalN : public signals::trackable,
                private noncopyable   // Exposition only
{
public:
  // types
  typedef typename Combiner::result_type result_type;
  typedef Combiner                       combiner_type;
  typedef Group                          group_type;
  typedef GroupCompare                   group_compare_type;
  typedef SlotFunction                   slot_function_type;
  typedef slot<SlotFunction>             slot_type;
  typedef unspecified                    slot_result_type;
  typedef unspecified                    slot_call_iterator;
  typedef T1                             argument_type;         // If N == 1
  typedef T1                             first_argument_type;   // If N == 2
  typedef T2                             second_argument_type;  // If N == 2
  typedef T1                             arg1_type;
  typedef T2                             arg2_type;
```

```
      .
      .
      .
  typedef TN                              argN_type;

  // static constants
  static const int arity = N;

  // construct/copy/destruct
  signalN(const combiner_type& = combiner_type(),
          const group_compare_type& = group_compare_type());
  ~signalN();

  // connection management
  signals::connection
  connect(const slot_type&, signals::connect_position = signals::at_back);
  signals::connection
  connect(const group_type&, const slot_type&,
          signals::connect_position = signals::at_back);
  void disconnect(const group_type&);
  template<typename Slot> void disconnect(const Slot&);
  void disconnect_all_slots();
  bool empty() const;
  std::size_t num_slots() const;

  // invocation
  result_type operator()(arg1_type, arg2_type, ..., argN_type);
  result_type operator()(arg1_type, arg2_type, ..., argN_type) const;

  // combiner access
  combiner_type& combiner();
  const combiner_type& combiner() const;
};
```

Description

The class template signalN covers several related classes signal0, signal1, signal2, etc., where the number suffix describes the number of function parameters the signal and its connected slots will take. Instead of enumerating all classes, a single pattern signalN will be described, where N represents the number of function parameters.

`signalN` construct/copy/destruct

1.
   ```
   signalN(const combiner_type& combiner = combiner_type(),
           const group_compare_type& compare = group_compare_type());
   ```

   Effects
   : Initializes the signal to contain no slots, copies the given combiner into internal storage, and stores the given group comparison function object to compare groups.

   Postconditions
   : `this->empty()`

2.
   ```
   ~signalN();
   ```

   Effects
   : Disconnects all slots connected to `*this`.

`signalN` connection management

1.
   ```
   signals::connection
   connect(const slot_type& slot,
           signals::connect_position at = signals::at_back);
   ```

```
signals::connection
connect(const group_type& group, const slot_type& slot,
        signals::connect_position at = signals::at_back);
```

Effects
: Connects the signal this to the incoming slot. If the slot is inactive, i.e., any of the trackable objects bound by the slot call have been destroyed, then the call to connect is a no–op. If the second version of `connect` is invoked, the slot is associated with the given group. The `at` parameter specifies where the slot should be connected: `at_front` indicates that the slot will be connected at the front of the list or group of slots and `at_back` indicates that the slot will be connected at the back of the list or group of slots.

Returns
: A `signals::connection` object that references the newly–created connection between the signal and the slot; if the slot is inactive, returns a disconnected connection.

Throws
: This routine meets the strong exception guarantee, where any exception thrown will cause the slot to not be connected to the signal.

Complexity
: Constant time when connecting a slot without a group name or logarithmic in the number of groups when connecting to a particular group.

Notes
: It is unspecified whether connecting a slot while the signal is calling will result in the slot being called immediately.

2.
```
void disconnect(const group_type& group);
template<typename Slot> void disconnect(const Slot& slot);
```

Effects
: If the parameter is (convertible to) a group name, any slots in the given group are disconnected. Otherwise, any slots equal to the given slot are disconnected.

Throws
: Will not throw unless a user destructor or equality operator == throws. If either throws, not all slots may be disconnected.

Complexity
: If a group is given, O(lg g) + k where g is the number of groups in the signal and k is the number of slots in the group. Otherwise, linear in the number of slots connected to the signal.

3.
```
void disconnect_all_slots();
```

Effects
: Disconnects all slots connected to the signal.

Postconditions
: `this->empty().`

Throws
: If disconnecting a slot causes an exception to be thrown, not all slots may be disconnected.

Complexity
: Linear in the number of slots known to the signal.

Notes
: May be called at any time within the lifetime of the signal, including during calls to the signal's slots.

4.
```
bool empty() const;
```

Returns
: `true` if no slots are connected to the signal, and `false` otherwise.

Throws
: Will not throw.

Complexity
: Linear in the number of slots known to the signal.

Rationale
: Slots can disconnect at any point in time, including while those same slots are being invoked. It is therefore possible that the implementation must search through a list of disconnected slots to determine if any slots

are still connected.

5.
```
std::size_t num_slots() const;
```

Returns
: The number of slots connected to the signal

Throws
: Will not throw.

Complexity
: Linear in the number of slots known to the signal.

Rationale
: Slots can disconnect at any point in time, including while those same slots are being invoked. It is therefore possible that the implementation must search through a list of disconnected slots to determine how many slots are still connected.

`signalN` invocation

1.
```
result_type operator()(arg1_type a1, arg2_type a2, ... , argN_type aN);
result_type operator()(arg1_type a1, arg2_type a2, ... , argN_type aN) const;
```

Effects
: Invokes the combiner with a `slot_call_iterator` range [first, last) corresponding to the sequence of calls to the slots connected to signal `*this`. Dereferencing an iterator in this range causes a slot call with the given set of parameters `(a1, a2, ..., aN)`, the result of which is returned from the iterator dereference operation.

Returns
: The result returned by the combiner.

Throws
: If an exception is thrown by a slot call, or if the combiner does not dereference any slot past some given slot, all slots after that slot in the internal list of connected slots will not be invoked.

Notes
: Only the slots associated with iterators that are actually dereferenced will be invoked. Multiple dereferences of the same iterator will not result in multiple slot invocations, because the return value of the slot will be cached.

  The `const` version of the function call operator will invoke the combiner as `const`, whereas the non−`const` version will invoke the combiner as non−`const`.

  Calling the function call operator may invoke undefined behavior if no slots are connected to the signal, depending on the combiner used. The default combiner is well−defined for zero slots when the return type is void but is undefined when the return type is any other type (because there is no way to synthesize a return value).

`signalN` combiner access

1.
```
combiner_type& combiner();
const combiner_type& combiner() const;
```

Returns
: A reference to the stored combiner.

Throws
: Will not throw.

Class template signal

boost::signal Safe multicast callback.

Synopsis

```
template<typename Signature,   // Function type R (T1, T2, ..., TN)
         typename Combiner = last_value<R>,
         typename Group = int,
         typename GroupCompare = std::less<Group>,
         typename SlotFunction = functionN<Signature> >
class signal : public signalN<R, T1, T2, ..., TN, Combiner, Group, GroupCompare, SlotFunction>
{
public:
  // construct/copy/destruct
  signal(const combiner_type& = combiner_type(),
         const group_compare_type& = group_compare_type());
};
```

Description

Class template signal is a thin wrapper around the numbered class templates signal0, signal1, etc. It accepts a function type with N arguments instead of N separate arguments, and derives from the appropriate signalN instantiation.

All functionality of this class template is in its base class signalN.

`signal` construct/copy/destruct

1.
```
signal(const combiner_type& combiner = combiner_type(),
       const group_compare_type& compare = group_compare_type());
```

   Effects
      Initializes the base class with the given combiner and comparison objects.

3.2. Header <boost/signals/slot.hpp>

```
namespace boost {
  template<typename SlotFunction> class slot;
}
```

Class template slot

boost::slot Pass slots as function arguments.

Synopsis

```
template<typename SlotFunction>
class slot {
public:
  // construct/copy/destruct
  template<typename Slot> slot(Slot);
};
```

Description

`slot` construct/copy/destruct

1. `template<typename Slot> slot(Slot target);`

   Effects
   : Invokes `visit_each` (unqualified) to discover pointers and references to `signals::trackable` objects in `target`.

     Initializes `this` to contain the incoming slot `target`, which may be any function object with which a `SlotFunction` can be constructed.

3.3. Header <boost/signals/trackable.hpp>

```
namespace boost {
  namespace signals {
    class trackable;
  }
}
```

Class trackable

boost::signals::trackable — Enables safe use of multicast callbacks.

Synopsis

```
class trackable {
public:
  // construct/copy/destruct
  trackable();
  trackable(const trackable&);
  trackable& operator=(const trackable&);
  ~trackable();
};
```

Description

The `trackable` class provides automatic disconnection of signals and slots when objects bound in slots (via pointer or reference) are destroyed. The `trackable` class may only be used as a public base class for some other class; when used as such, that class may be bound to function objects used as part of slots. The manner in which a `trackable` object tracks the set of signal–slot connections it is a part of is unspecified.

The actual use of `trackable` is contingent on the presence of appropriate visit_each overloads for any type that may contain pointers or references to trackable objects.

`trackable` construct/copy/destruct

1. `trackable();`

   Effects
   : Sets the list of connected slots to empty.

   Throws
   : Will not throw.
2. `trackable(const trackable& other);`

   Effects
   : Sets the list of connected slots to empty.

   Throws

Will not throw.

Rationale

Signal–slot connections can only be created via calls to an explicit connect method, and therefore cannot be created here when trackable objects are copied.

3. `trackable& operator=(const trackable& other);`

Effects

Sets the list of connected slots to empty.

Returns

`*this`

Throws

Will not throw.

Rationale

Signal–slot connections can only be created via calls to an explicit connect method, and therefore cannot be created here when trackable objects are copied.

4. `~trackable();`

Effects

Disconnects all signal/slot connections that contain a pointer or reference to this trackable object that can be found by visit_each.

3.4. Header <boost/signals/connection.hpp>

```
namespace boost {
  namespace signals {
    class connection;
    void swap(connection&, connection&);
    class scoped_connection;
  }
}
```

Class connection

boost::signals::connection   Query/disconnect a signal–slot connection.

Synopsis

```
class connection {
public:
  // construct/copy/destruct
  connection();
  connection(const connection&);
  connection& operator=(const connection&);

  // connection management
  void disconnect() const;
  bool connected() const;

  // modifiers
  void swap(const connection&);

  // comparisons
  bool operator==(const connection&) const;
  bool operator<(const connection&) const;
};

// specialized algorithms
void swap(connection&, connection&);
```

Description

The connection class represents a connection between a Signal and a Slot. It is a lightweight object that has the ability to query whether the signal and slot are currently connected, and to disconnect the signal and slot. It is always safe to query or disconnect a connection.

`connection` construct/copy/destruct

1. `connection();`

   Effects
   : Sets the currently represented connection to the NULL connection.

   Postconditions
   : `!this->connected()`.

   Throws
   : Will not throw.

2. `connection(const connection& other);`

   Effects
   : `this` references the connection referenced by `other`.

   Throws
   : Will not throw.

3. `connection& operator=(const connection& other);`

   Effects
   : `this` references the connection referenced by `other`.

   Throws
   : Will not throw.

`connection` connection management

1. `void disconnect() const;`

   Effects
   : If `this->connected()`, disconnects the signal and slot referenced by this; otherwise, this operation is a no–op.

   Postconditions
   : `!this->connected()`.

2. `bool connected() const;`

   Returns
   : `true` if this references a non–NULL connection that is still active (connected), and `false` otherwise.

   Throws
   : Will not throw.

`connection` modifiers

1. `void swap(const connection& other);`

   Effects
   : Swaps the connections referenced in `this` and `other`.

   Throws
   : Will not throw.

`connection` comparisons

1. `bool operator==(const connection& other) const;`

   Returns
   : `true` if `this` and `other` reference the same connection or both reference the NULL connection, and `false` otherwise.

   Throws
   : Will not throw.

2. `bool operator<(const connection& other) const;`

   Returns
   : `true` if the connection referenced by `this` precedes the connection referenced by `other` based on some unspecified ordering, and `false` otherwise.

   Throws
   : Will not throw.

`connection` specialized algorithms

1. `void swap(connection& x, connection& y);`

   Effects
   : `x.swap(y)`

   Throws
   : Will not throw.

Class scoped_connection

boost::signals::scoped_connection — Limits a signal–slot connection lifetime to a particular scope.

Synopsis

```
class scoped_connection : private noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  scoped_connection(const connection&);
  ~scoped_connection();

  // connection management
  void disconnect() const;
  bool connected() const;
};
```

Description

`scoped_connection` construct/copy/destruct

1. `scoped_connection(const connection& other);`

   Effects
   : `this` references the connection referenced by `other`.

   Throws
   : Will not throw.

2. `~scoped_connection();`

Effects

If `this->connected()`, disconnects the signal–slot connection.

`scoped_connection` connection management

1. `void disconnect() const;`

   Effects

   If `this->connected()`, disconnects the signal and slot referenced by this; otherwise, this operation is a no–op.

   Postconditions

   `!this->connected()`.
2. `bool connected() const;`

   Returns

   `true` if this references a non–NULL connection that is still active (connected), and `false` otherwise.

   Throws

   Will not throw.

3.5. Header <boost/visit_each.hpp>

```
namespace boost {
  template<typename Visitor, typename T>
    void visit_each(const Visitor&, const T&, int);
}
```

Function template visit_each

boost::visit_each — Allow limited exploration of class members.

Synopsis

```
template<typename Visitor, typename T>
  void visit_each(const Visitor& visitor, const T& t, int );
```

Description

The visit_each mechanism allows a visitor to be applied to every subobject in a given object. It is used by the Signals library to discover signals::trackable objects within a function object, but other uses may surface if used universally (e.g., conservative garbage collection). To fit within the visit_each framework, a visit_each overload must be supplied for each object type.

Effects

`visitor(t)`, and for every subobject `x` of `t`:

- If `x` is a reference, `visit_each(visitor, ref(x), 0)`
- Otherwise, `visit_each(visitor, x, 0)`

Notes

The third parameter is `long` for the fallback version of visit_each and the argument supplied to this third paramter must always be 0. The third parameter is an artifact of the widespread lack of proper function template ordering, and will be removed in the future.

Library authors will be expected to add additional overloads that specialize the T argument for their classes, so that subobjects can be visited.

Calls to visit_each are required to be unqualified, to enable argument−dependent lookup.

3.6. Header <boost/last_value.hpp>

```
namespace boost {
  template<typename T> class last_value;

  template<> class last_value<void>;
}
```

Class template last_value

boost::last_value   Evaluate an InputIterator sequence and return the last value in the sequence.

Synopsis

```
template<typename T>
class last_value {
public:
  // types
  typedef T result_type;

  // invocation
  template<typename InputIterator>
    result_type operator()(InputIterator, InputIterator) const;
};
```

Description

`last_value` invocation

1.
   ```
   template<typename InputIterator>
     result_type operator()(InputIterator first, InputIterator last) const;
   ```

   Requires

   `first != last`

   Effects

   Dereferences every iterator in the sequence `[first, last)`.

   Returns

   The result of dereferencing the iterator `last-1`.

Specializations

- Class last_value<void>

Class last_value<void>

boost::last_value<void>   Evaluate an InputIterator sequence.

Synopsis

```
class last_value<void> {
public:
  // types
  typedef unspecified result_type;
```

```
  // invocation
  template<typename InputIterator>
    result_type operator()(InputIterator, InputIterator) const;
};
```

Description

`last_value` invocation

1.
```
template<typename InputIterator>
  result_type operator()(InputIterator first, InputIterator last) const;
```

   Effects

   Dereferences every iterator in the sequence `[first, last)`.

4. Frequently Asked Questions

4.1. Don't noncopyable signal semantics mean that a class with a signal member will be noncopyable as well?

No. The compiler will not be able to generate a copy constructor or copy assignment operator for your class if it has a signal as a member, but you are free to write your own copy constructor and/or copy assignment operator. Just don't try to copy the signal.

4.2. Is Boost.Signals thread−safe?

No. Using Boost.Signals in a multithreaded concept is very dangerous, and it is very likely that the results will be less than satisfying. Boost.Signals will support thread safety in the future.

4.3. How do I get Boost.Signals to work with Qt?

When building with Qt, the Moc keywords `signals` and `slots` are defined using preprocessor macros, causing programs using Boost.Signals and Qt together to fail to compile. Although this is a problem with Qt and not Boost.Signals, a user can use the two systems together by defining the `BOOST_SIGNALS_NAMESPACE` macro to some other identifier (e.g., `signalslib`) when building and using the Boost.Signals library. Then the namespace of the Boost.Signals library will be `boost::BOOST_SIGNALS_NAMESPACE` instead of `boost::signals`. To retain the original namespace name in translation units that do not interact with Qt, you can use a namespace alias:

```
namespace boost {
  namespace signals = BOOST_SIGNALS_NAMESPACE;
}
```

5. Design Overview

5.1. Type Erasure

"Type erasure", where static type information is eliminated by the use of dynamically dispatched interfaces, is used extensively within the Boost.Signals library to reduce the amount of code generated by template instantiation. Each signal must manage a list of slots and their associated connections, along with a `std::map` to map from group identifiers to their associated connections. However, instantiating this map for every token type, and perhaps within each translation unit (for some popular template instantiation strategies) increase compile time overhead and space overhead.

To combat this so−called "template bloat", we use Boost.Function and Boost.Any to store unknown types and operations. Then, all of the code for handling the list of slots and the mapping from slot identifiers to connections is factored into the class `signal_base` that deals exclusively with the `any` and `function` objects, hiding the actual implementations using the well−known pimpl idiom. The actual `signalN` class templates deal only with code that will change depending on the number of arguments or which is inherently template−dependent (such as connection).

5.2. `connection` class

The `connection` class is central to the behavior of the Boost.Signals library. It is the only entity within the Boost.Signals system that has knowledge of all objects that are associated by a given connection. To be specific, the `connection` class itself is merely a thin wrapper over a `shared_ptr` to a `basic_connection` object.

`connection` objects are stored by all participants in the Signals system: each `trackable` object contains a list of `connection` objects describing all connections it is a part of; similarly, all signals contain a set of pairs that define a slot. The pairs consist of a slot function object (generally a Boost.Function object) and a `connection` object (that will disconnect on destruction). Finally, the mapping from slot groups to slots is based on the key value in a `std::multimap` (the stored data in the `std::multimap` is the slot pair).

5.3. Slot Call Iterator

The slot call iterator is conceptually a stack of iterator adaptors that modify the behavior of the underlying iterator through the list of slots. The following table describes the type and behavior of each iterator adaptor required. Note that this is only a conceptual model: the implementation collapses all these layers into a single iterator adaptor because several popular compilers failed to compile the implementation of the conceptual model.

Iterator Adaptor	**Purpose**
Slot List Iterator	An iterator through the list of slots connected to a signal. The `value_type` of this iterator will be `std::pair<any, connection>`, where the `any` contains an instance of the slot function type.
Filter Iterator Adaptor	This filtering iterator adaptor filters out slots that have been disconnected, so we never see a disconnected slot in later stages.
Projection Iterator Adaptor	The projection iterator adaptor returns a reference to the first member of the pair that constitutes a connected slot (e.g., just the `boost::any` object that holds the slot function).
Transform Iterator Adaptor	This transform iterator adaptor performs an `any_cast` to extract a reference to the slot function with the appropriate slot function type.
Transform Iterator Adaptor	This transform iterator adaptor calls the function object returned by dereferencing the underlying iterator with the set of arguments given to the signal itself, and returns the result of that slot call.
Input Caching Iterator Adaptor	This iterator adaptor caches the result of dereferencing the underlying iterator. Therefore, dereferencing this iterator multiple times will only result in the underlying iterator being dereferenced once; thus, a slot can only be called once but its result can be used multiple times.
Slot Call Iterator	Iterates over calls to each slot.

5.4. `visit_each` function template

The `visit_each` function template is a mechanism for discovering objects that are stored within another object. Function template `visit_each` takes three arguments: an object to explore, a visitor function object that is invoked with each subobject, and the `int` 0.

The third parameter is merely a temporary solution to the widespread lack of proper function template partial ordering. The primary `visit_each` function template specifies this third parameter type to be `long`, whereas any user specializations must specify their third parameter to be of type `int`. Thus, even though a broken compiler cannot tell the ordering between, e.g., a match against a parameter `T` and a parameter `A<T>`, it can determine that the conversion from the integer 0 to `int` is better than the conversion to `long`. The ordering determined by this conversion thus achieves partial ordering of the function templates in a limited, but successful, way. The following example illustrates the use of this technique:

```
template<typename> class A {};
```

```
template<typename T> void foo(T, long);
template<typename T> void foo(A<T>, int);
A<T> at;
foo(at, 0);
```

In this example, we assume that our compiler can not tell that `A<T>` is a better match than `T`, and therefore assume that the function templates cannot be ordered based on that parameter. Then the conversion from 0 to `int` is better than the conversion from 0 to `long`, and the second function template is chosen.

6. Design Rationale

6.1. Choice of Slot Definitions

The definition of a slot differs amongst signals and slots libraries. Within Boost.Signals, a slot is defined in a very loose manner: it can be any function object that is callable given parameters of the types specified by the signal, and whose return value is convertible to the result type expected by the signal. However, alternative definitions have associated pros and cons that were considered prior to the construction of Boost.Signals.

- **Slots derive from a specific base class**: generally a scheme such as this will require all user−defined slots to derive from some library−specified `Slot` abstract class that defines a virtual function calling the slot. Adaptors can be used to convert a definition such as this to a definition similar to that used by Boost.Signals, but the use of a large number of small adaptor classes containing virtual functions has been found to cause an unacceptable increase in the size of executables (polymorphic class types require more code than non−polymorphic types).

  This approach does have the benefit of simplicity of implementation and user interface, from an object−oriented perspective.
- **Slots constructed from a set of primitives**: in this scheme the slot can have a limited set of types (often derived from a common abstract base class) that are constructed from some library−defined set of primitives that often include conversions from free function pointers and member function pointers, and a limited set of binding capabilities. Such an approach is reasonably simple and cover most common cases, but it does not allow a large degree of flexibility in slot construction. Libraries for function object composition have become quite advanced and it is out of the scope of a signals and slots library to encorporate such enhancements. Thus Boost.Signals does not include argument binding or function object composition primitives, but instead provides a hook (via the `visit_each` mechanism) that allows existing binder/composition libraries to provide the necessary information to Signals.

Users not satisfied with the slot definition choice may opt to replace the default slot function type with an alternative that meets their specific needs.

6.2. User−level Connection Management

Users need to have fine control over the connection of signals to slots and their eventual disconnection. The approach taken by Boost.Signals is to return a `connection` object that enables connected/disconnected query, manual disconnection, and an automatic disconnection on destruction mode. Some other possible interfaces include:

- **Pass slot to disconnect**: in this interface model, the disconnection of a slot connected with `sig.connect(slot)` is performed via `sig.disconnect(slot)`. Internally, a linear search using slot comparison is performed and the slot, if found, is removed from the list. Unfortunately, querying connectedness will generally also end up as linear−time operations. This model also fails for implementation reasons when slots become more complex than simple function pointers, member function pointers and a limited set of compositions and argument binders: to match the slot given in the call to `disconnect` with an existing slot we would need to be able to compare arbitrary function objects, which is not feasible.
- **Pass a token to disconnect**: this approach identifies slots with a token that is easily comparable (e.g., a string), enabling slots to be arbitrary function objects. While this approach is essentially equivalent to the approach taken by Boost.Signals, it is possibly more error−prone for several reasons:

- Connections and disconnections must be paired, so the problem becomes similar to the problems incurred when pairing `new` and `delete` for dynamic memory allocation. While errors of this sort would not be catastrophic for a signals and slots implementation, their detection is generally nontrivial.
- Tokens must be unique, otherwise two slots will have the same name and will be indistinguishable. In environments where many connections will be made dynamically, name generation becomes an additional task for the user. Uniqueness of tokens also results in an additional failure mode when attempting to connect a slot using a token that has already been used.
- More parameterization would be required, because the token type must be user−defined. Additional parameterization steepens the learning curver and overcomplicates a simple interface.

This type of interface is supported in Boost.Signals via the slot grouping mechanism. It augments the `connection` object−based connection management scheme.

6.3. Combiner Interface

The Combiner interface was chosen to mimic a call to an algorithm in the C++ standard library. It is felt that by viewing slot call results as merely a sequence of values accessed by input iterators, the combiner interface would be most natural to a proficient C++ programmer. Competing interface design generally required the combiners to be constructed to conform to an interface that would be customized for (and limited to) the Signals library. While these interfaces are generally enable more straighforward implementation of the signals & slots libraries, the combiners are unfortunately not reusable (either in other signals & slots libraries or within other generic algorithms), and the learning curve is steepened slightly to learn the specific combiner interface.

The Signals formulation of combiners is based on the combiner using the "pull" mode of communication, instead of the more complex "push" mechanism. With a "pull" mechanism, the combiner's state can be kept on the stack and in the program counter, because whenever new data is required (i.e., calling the next slot to retrieve its return value), there is a simple interface to retrieve that data immediately and without returning from the combiner's code. Contrast this with the "push" mechanism, where the combiner must keep all state in class members because the combiner's routines will be invoked for each signal called. Compare, for example, a combiner that returns the maximum element from calling the slots. If the maximum element ever exceeds 100, no more slots are to be called.

Pull

```
struct pull_max {
  typedef int result_type;

  template<typename InputIterator>
  result_type operator()(InputIterator first,
                         InputIterator last)
  {
    if (first == last)
      throw std::runtime_error("Empty!");

    int max_value = *first++;
    while(first != last && *first <= 100) {
      if (*first > max_value)
        max_value = *first;
      ++first;
    }

    return max_value;
  }
};
```

Push

```
struct push_max {
  typedef int result_type;

  push_max() : max_value(), got_first(false) {}

  // returns false when we want to stop
  bool operator()(int result) {
    if (result > 100)
      return false;

    if (!got_first) {
      got_first = true;
      max_value = result;
      return true;
    }

    if (result > max_value)
      max_value = result;

    return true;
  }

  int get_value() const
  {
    if (!got_first)
      throw std::runtime_error("Empty!");
    return max_value;
  }
```

	`private:`
	`  int  max_value;`
	`  bool got_first;`
	`};`

There are several points to note in these examples. The "pull" version is a reusable function object that is based on an input iterator sequence with an integer `value_type`, and is very straightforward in design. The "push" model, on the other hand, relies on an interface specific to the caller and is not generally reusable. It also requires extra state values to determine, for instance, if any elements have been received. Though code quality and ease–of–use is generally subjective, the "pull" model is clearly shorter and more reusable and will often be construed as easier to write and understand, even outside the context of a signals & slots library.

The cost of the "pull" combiner interface is paid in the implementation of the Signals library itself. To correctly handle slot disconnections during calls (e.g., when the dereference operator is invoked), one must construct the iterator to skip over disconnected slots. Additionally, the iterator must carry with it the set of arguments to pass to each slot (although a reference to a structure containing those arguments suffices), and must cache the result of calling the slot so that multiple dereferences don't result in multiple calls. This apparently requires a large degree of overhead, though if one considers the entire process of invoking slots one sees that the overhead is nearly equivalent to that in the "push" model, but we have inverted the control structures to make iteration and dereference complex (instead of making combiner state–finding complex).

6.4. Connection Interfaces: += operator

Boost.Signals supports a connection syntax with the form `sig.connect(slot)`, but a more terse syntax `sig += slot` has been suggested (and has been used by other signals & slots implementations). There are several reasons as to why this syntax has been rejected:

- **It's unnecessary**: the connection syntax supplied by Boost.Signals is no less powerful that that supplied by the += operator. The savings in typing (`connect()` vs. +=) is essentially negligible. Furthermore, one could argue that calling `connect()` is more readable than an overload of +=.
- **Ambiguous return type**: there is an ambiguity concerning the return value of the += operation: should it be a reference to the signal itself, to enable `sig += slot1 += slot2`, or should it return a `connection` for the newly–created signal/slot connection?
- **Gateway to operators –=, +**: when one has added a connection operator +=, it seems natural to have a disconnection operator –=. However, this presents problems when the library allows arbitrary function objects to implicitly become slots, because slots are no longer comparable.

  The second obvious addition when one has `operator+=` would be to add a + operator that supports addition of multiple slots, followed by assignment to a signal. However, this would require implementing + such that it can accept any two function objects, which is technically infeasible.

6.5. `trackable` rationale

The `trackable` class is the primary user interface to automatic connection lifetime management, and its design affects users directly. Two issues stick out most: the odd copying behavior of `trackable`, and the limitation requiring users to derive from `trackable` to create types that can participate in automatic connection management.

6.5.1. `trackable` copying behavior

The copying behavior of `trackable` is essentially that `trackable` subobjects are never copied; instead, the copy operation is merely a no–op. To understand this, we look at the nature of a signal–slot connection and note that the connection is based on the entities that are being connected; when one of the entities is destroyed, the connection is destroyed. Therefore, when a `trackable` subobject is copied, we cannot copy the connections because the connections don't refer to the target entity – they refer to the source entity. This reason is dual to the reason signals are noncopyable: the slots connected to them are connected to that particular signal, not the data contained in the signal.

6.5.2. Why derivation from `trackable`?

For `trackable` to work properly, there are two constraints:

- `trackable` must have storage space to keep track of all connections made to this object.
- `trackable` must be notified when the object is being destructed so that it can disconnect its connections.

Clearly, deriving from `trackable` meets these two guidelines. We have not yet found a superior solution.

6.6. Comparison with other Signal/Slot implementations

6.6.1. libsigc++

libsigc++ is a C++ signals & slots library that originally started as part of an initiative to wrap the C interfaces to GTK libraries in C++, and has grown to be a separate library maintained by Karl Nelson. There are many similarities between libsigc++ and Boost.Signals, and indeed Boost.Signals was strongly influenced by Karl Nelson and libsigc++. A cursory inspection of each library will find a similar syntax for the construction of signals and in the use of connections and automatic connection lifetime management. There are some major differences in design that separate these libraries:

- **Slot definitions**: slots in libsigc++ are created using a set of primitives defined by the library. These primitives allow binding of objects (as part of the library), explicit adaptation from the argument and return types of the signal to the argument and return types of the slot (libsigc++ is, by default, more strict about types than Boost.Signals). A discussion of this approach with a comparison against the approach taken by Boost.Signals is given in Choice of Slot Definitions.
- **Combiner/Marshaller interface**: the equivalent to Boost.Signals combiners in libsigc++ are the marshallers. Marshallers are similar to the "push" interface described in Combiner Interface, and a proper treatment of the topic is given there.

6.6.2. .NET delegates

Microsoft has introduced the .NET Framework and an associated set of languages and language extensions, one of which is the delegate. Delegates are similar to signals and slots, but they are more limited than most C++ signals and slots implementations in that they:

- Require exact type matches between a delegate and what it is calling.
- Only return the result of the last target called, with no option for customization.
- Must call a method with `this` already bound.

7. Testsuite

7.1. Acceptance tests

Test	Type	Description	If failing...
dead_slot_test.cpp	run	Ensure that calling connect with a slot that has already been disconnected via deletion does not actually connect to the slot.	
deletion_test.cpp	run	Test deletion of slots.	
ordering_test.cpp	run	Test slot group ordering.	
signal_n_test.cpp	run	Basic test of signal/slot connections and invocation using the boost::signalN class templates.	

signal_test.cpp	run	Basic test of signal/slot connections and invocation using the boost::signal class template.	The boost::signal class template may not be usable on your compiler. However, the boost::signalN class templates may still be usable.
trackable_test.cpp	run	Test automatic lifetime management using boost::trackable objects.	

Chapter 8. Boost String Algorithms Library

Pavol Droba

1. Introduction

The String Algorithm Library provides a generic implementation of string–related algorithms which are missing in STL. It is an extension to the algorithms library of STL and it includes trimming, case conversion, predicates and find/replace functions. All of them come in different variants so it is easier to choose the best fit for a particular need.

The implementation is not restricted to work with a particular container (like `std::basic_string`), rather it is as generic as possible. This generalization is not compromising the performance since algorithms are using container specific features when it means a performance gain.

Important note: In this documentation we use term *string* to designate a sequence of *characters* stored in an arbitrary container. A *string* is not restricted to `std::basic_string` and *character* does not have to be `char` or `wchar_t`, although these are most common candidates. Consult the design chapter to see precise specification of supported string types.

The library interface functions and classes are defined in namespace `boost::algorithm`, and they are lifted into namespace `boost` via using declaration.

The documentation is divided into several sections. For a quick start read the Usage section followed by Quick Reference. The Design Topics, Concepts and Rationale provide some explanation about the library design and structure an explain how it should be used. See the Reference for the complete list of provided utilities and algorithms. Functions and classes in the reference are organized by the headers in which they are defined. The reference contains links to the detailed description for every entity in the library.

2. Usage

2.1. First Example

Using the algorithms is straightforward. Let us have a look at the first example:

```
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost;

// ...

string str1(" hello world! ");
to_upper(str1);  // str1 == " HELLO WORLD! "
trim(str1);      // str1 == "HELLO WORLD!"

string str2=
   to_lower_copy(
      ireplace_first_copy(
         str1,"hello","goodbye")); // str2 == "goodbye world!"
```

This example converts str1 to upper case and trims spaces from the start and the end of the string. str2 is then created as a copy of str1 with "hello" replaced with "goodbye". This example demonstrates several important concepts used in the library:

- **Container parameters:** Unlike in the STL algorithms, parameters are not specified only in the form of iterators. The STL convention allows for great flexibility, but it has several limitations. It is not possible to *stack* algorithms together, because a container is passed in two parameters. Therefore it is not possible to use a return value from another algorithm. It is considerably easier to write `to_lower(str1)`, than `to_lower(str1.begin(), str1.end())`.

  The magic of collection_traits provides a uniform way of handling different string types. If there is a need to pass a pair of iterators, `iterator_range` can be used to package iterators into a structure with a compatible interface.
- **Copy vs. Mutable:** Many algorithms in the library are performing a transformation of the input. The transformation can be done in−place, mutating the input sequence, or a copy of the transformed input can be created, leaving the input intact. None of these possibilities is superior to the other one and both have different advantages and disadvantages. For this reason, both are provided with the library.
- **Algorithm stacking:** Copy versions return a transformed input as a result, thus allow a simple chaining of transformations within one expression (i.e. one can write `trim_copy(to_upper_copy(s))`. Mutable versions have `void` return, to avoid misuse.
- **Naming:** Naming follows the conventions from the Standard C++ Library. If there is a copy and a mutable version of the same algorithm, the mutable version has no suffix and the copy version has the suffix *_copy*. Some algorithms have the prefix *i* (e.g. ifind_first()). This prefix identifies that the algorithm works in a case−insensitive manner.

To use the library, include the boost/algorithm/string.hpp header. If the regex related functions are needed, include the boost/algorithm/string_regex.hpp header.

2.2. Case conversion

STL has a nice way of converting character case. Unfortunately, it works only for a single character and we want to convert a string,

```
string str1("HeLlO WoRld!");
to_upper(str1); // str1=="HELLO WORLD!"
```

to_upper() and to_lower() convert the case of characters in a string using a specified locale.

For more information see the reference for boost/algorithm/string/case_conv.hpp.

2.3. Predicates and Classification

A part of the library deals with string related predicates. Consider this example:

```
bool is_executable( string& filename )
{
    return
        iends_with(filename, ".exe") ||
        iends_with(filename, ".com");
}

// ...
string str1("command.com");
cout
    << str1
    << is_executable("command.com")? "is": "is not"
    << "an executable"
    << endl; // prints "command.com is an executable"

//..
char text1[]="hello world!";
cout
```

```
        << text1
        << all( text1, is_lower() )? "is": "is not"
        << " written in the lower case"
        << endl; // prints "hello world! is written in the lower case"
```

The predicates determine whether if a substring is contained in the input string under various conditions. The conditions are: a string starts with the substring, ends with the substring, simply contains the substring or if both strings are equal. See the reference for boost/algorithm/string/predicate.hpp for more details.

In addition the algorithm all() checks all elements of a container to satisfy a condition specified by a predicate. This predicate can be any unary predicate, but the library provides a bunch of useful string−related predicates and combinators ready for use. These are located in the boost/algorithm/string/classification.hpp header. Classification predicates can be combined using logical combinators to form a more complex expressions. For example: `is_from_range('a','z') || is_digit()`

2.4. Trimming

When parsing the input from a user, strings usually have unwanted leading or trailing characters. To get rid of them, we need trim functions:

```
    string str1="     hello world!     ";
    string str2=trim_left_copy(str1);   // str2 == "hello world!     "
    string str3=trim_right_copy(str2);  // str3 == "     hello world!"
    trim(str1);                         // str1 == "hello world!"

    string phone="00423333444";
    // remove leading 0 from the phone number
    trim_left_if(phone,is_any_of("0")); // phone == "423333444"
```

It is possible to trim the spaces on the right, on the left or on both sides of a string. And for those cases when there is a need to remove something else than blank space, there are *_if* variants. Using these, a user can specify a functor which will select the *space* to be removed. It is possible to use classification predicates like is_digit() mentioned in the previous paragraph. See the reference for the boost/algorithm/string/trim.hpp.

2.5. Find algorithms

The library contains a set of find algorithms. Here is an example:

```
    char text[]="hello dolly!";
    iterator_range<char*> result=find_last(text,"ll");

    transform( result.begin(), result.end(), result.begin(), bind2nd(plus<char>(), 1) );
    // text = "hello dommy!"

    to_upper(result); // text == "hello doMMy!"

    // iterator_range is convertible to bool
    if(find_first(text, "dolly"))
    {
        cout << "Dolly is there" << endl;
    }
```

We have used find_last() to search the `text` for "ll". The result is given in the `iterator_range`. This range delimits the part of the input which satisfies the find criteria. In our example it is the last occurrence of "ll". As we can see, input of the find_last() algorithm can be also char[] because this type is supported by collection_traits. The following lines transform the result. Notice that `iterator_range` has familiar `begin()` and `end()` methods, so it can be used like any other STL container. Also it is convertible to bool therefore it is easy to use find algorithms for a simple containment checking.

Find algorithms are located in boost/algorithm/string/find.hpp.

2.6. Replace Algorithms

Find algorithms can be used for searching for a specific part of string. Replace goes one step further. After a matching part is found, it is substituted with something else. The substitution is computed from the original, using some transformation.

```
string str1="Hello  Dolly,   Hello World!"
replace_first(str1, "Dolly", "Jane");      // str1 == "Hello  Jane,   Hello World!"
replace_last(str1, "Hello", "Goodbye");    // str1 == "Hello  Jane,   Goodbye World!"
erase_all(str1, " ");                      // str1 == "HelloJane,GoodbyeWorld!"
erase_head(str1, 6);                       // str1 == "Jane,GoodbyeWorld!"
```

For the complete list of replace and erase functions see the reference. There is a lot of predefined function for common usage, however, the library allows you to define a custom `replace()` that suits a specific need. There is a generic find_format() function which takes two parameters. The first one is a Finder object, the second one is a Formatter object. The Finder object is a functor which performs the searching for the replacement part. The Formatter object takes the result of the Finder (usually a reference to the found substring) and creates a substitute for it. Replace algorithm puts these two together and makes the desired substitution.

Check boost/algorithm/string/replace.hpp, boost/algorithm/string/erase.hpp and boost/algorithm/string/find_format.hpp for reference.

2.7. Find Iterator

An extension to find algorithms it the Find Iterator. Instead of searching for just a one part of a string, the find iterator allows us to iterate over the substrings matching the specified criteria. This facility is using the Finder to incrementally search the string. Dereferencing a find iterator yields an `iterator_range` object, that delimits the current match.

There are two iterators provided find_iterator and split_iterator. The former iterates over substrings that are found using the specified Finder. The latter iterates over the gaps between these substrings.

```
string str1("abc-*-ABC-*-aBc");
// Find all 'abc' substrings (ignoring the case)
// Create a find_iterator
typedef find_iterator<string::iterator> string_find_iterator;
for(string_find_iterator It=
        make_find_iterator(str1, first_finder("abc", is_iequal()));
    It!=string_find_iterator();
    ++It)
{
    cout << copy_iterator_range<std::string>(*It) << endl;
}

// Output will be:
// abc
// ABC
// aBC

typedef split_iterator<string::iterator> string_split_iterator;
for(string_find_iterator It=
    make_split_iterator(str1, first_finder("-*-", is_iequal()));
    It!=string_find_iterator();
    ++It)
{
    cout << copy_iterator_range<std::string>(*It) << endl;
}

// Output will be:
// abc
// ABC
```

```
    // aBC
```

Note that the find iterators have only one template parameter. It is the base iterator type. The Finder is specified at runtime. This allows us to typedef a find iterator for common string types and reuse it. Additionally make_*_iterator functions help to construct a find iterator for a particular collection.

See the reference in boost/algorithm/string/find_iterator.hpp.

2.8. Split

Split algorithms are an extension to the find iterator for one common usage scenario. These algorithms use a find iterator and store all matches into the provided container. This container must be able to hold copies (e.g. `std::string`) or references (e.g. `iterator_range`) of the extracted substrings.

Two algorithms are provided. find_all() finds all copies of a string in the input. split() splits the input into parts.

```
    string str1("hello abc-*-ABC-*-aBc goodbye");

    typedef vector< iterator_range<string::iterator> > find_vector_type;

    find_vector_type FindVec; // #1: Search for separators
    ifind_all( FindVec, str1, "abc" ); // FindVec == { [abc],[ABC],[aBc] }

    typedef vector< string > split_vector_type;

    split_vector_type SplitVec; // #2: Search for tokens
    split( SplitVec, str1, is_any_of("-*") ); // SplitVec == { "hello abc","ABC","aBc goodbye" }
```

`[hello]` designates an `iterator_range` delimiting this substring.

First example show how to construct a container to hold references to all extracted substrings. Algorithm ifind_all() puts into FindVec references to all substrings that are in case−insensitive manner equal to "abc".

Second example uses split() to split string str1 into parts separated by characters '−' or '*'. These parts are then put into the SplitVec. It is possible to specify if adjacent separators are concatenated or not.

More information can be found in the reference: boost/algorithm/string/split.hpp.

3. Quick Reference

3.1. Algorithms

Table 8.1. Case Conversion

Algorithm name	Description	Functions

`to_upper`	Convert a string to upper case	to_upper_copy()
		to_upper()

`to_lower`	Convert a string to lower case	to_lower_copy()
		to_lower()

Table 8.2. Trimming

Algorithm name	Description	Functions

`trim_left`	Remove leading spaces from a string	trim_left_copy_if()
		trim_left_if()
		trim_left_copy()
		trim_left()

`trim_right`	Remove trailing spaces from a string	trim_right_copy_if()
		trim_right_if()
		trim_right_copy()
		trim_right()

`trim`	Remove leading and trailing spaces from a string	trim_copy_if()
		trim_if()
		trim_copy()
		trim()

Table 8.3. Predicates

Algorithm name	Description	Functions

`starts_with`	Check if a string is a prefix of the other one	starts_with()
		istarts_with()

`ends_with`	Check if a string is a suffix of the other one	ends_with()
		iends_with()

`contains`	Check if a string is contained of the other one	contains()
		icontains()

`equals`	Check if two strings are equal	equals()
		iequals()

`all`	Check if all elements of a string satisfy the given predicate	all()

Table 8.4. Find algorithms

Algorithm name	Description	Functions

find_first	Find the first occurrence of a string in the input	find_first()
		ifind_first()

find_last	Find the last occurrence of a string in the input	find_last()
		ifind_last()

find_nth	Find the nth (zero–indexed) occurrence of a string in the input	find_nth()
		ifind_nth()

find_head	Retrieve the head of a string	find_head()
find_tail	Retrieve the tail of a string	find_tail()
find_token	Find first matching token in the string	find_token()
find_regex	Use the regular expression to search the string	find_regex()

find	Generic find algorithm	find()

Table 8.5. Erase/Replace

Algorithm name	Description	Functions

replace/erase_first	Replace/Erase the first occurrence of a string in the input	replace_first()
		replace_first_copy()
		ireplace_first()
		ireplace_first_copy()
		erase_first()
		erase_first_copy()
		ierase_first()
		ierase_first_copy()

replace/erase_last	Replace/Erase the last occurrence of a string in the input	replace_last()
		replace_last_copy()
		ireplace_last()
		ireplace_last_copy()
		erase_last()
		erase_last_copy()
		ierase_last()
		ierase_last_copy()

replace/erase_nth	Replace/Erase the nth (zero−indexed) occurrence of a string in the input	replace_nth()
		replace_nth_copy()
		ireplace_nth()
		ireplace_nth_copy()
		erase_nth()
		erase_nth_copy()
		ierase_nth()
		ierase_nth_copy()

replace/erase_all	Replace/Erase the all occurrences of a string in the input	replace_all()
		replace_all_copy()
		ireplace_all()
		ireplace_all_copy()
		erase_all()
		erase_all_copy()
		ierase_all()
		ierase_all_copy()

replace/erase_head	Replace/Erase the head of the input	replace_head()
		replace_head_copy()
		erase_head()
		erase_head_copy()

replace/erase_tail	Replace/Erase the tail of the input	replace_tail()
		replace_tail_copy()
		erase_tail()
		erase_tail_copy()

replace/erase_regex	Replace/Erase a substring matching the given regular expression	replace_regex()
		replace_regex_copy()
		erase_regex()
		erase_regex_copy()

replace/erase_regex_all	Replace/Erase all substrings matching the given regular expression	replace_all_regex()
		replace_all_regex_copy()
		erase_all_regex()
		erase_all_regex_copy()

find_format	Generic replace algorithm	find_format()
		find_format_copy()
		find_format_all()
		find_format_all_copy()()

Table 8.6. Split

Algorithm name	Description	Functions

find_all	Find/Extract all matching substrings in the input	find_all()
		ifind_all()
		find_all_regex()

split	Split input into parts	split()
		split_regex()

3.2. Finders and Formatters

Table 8.7. Finders

Finder	Description	Generators
first_finder	Search for the first match of the string in an input	first_finder()
last_finder	Search for the last match of the string in an input	last_finder()
nth_finder	Search for the nth (zero−indexed) match of the string in an input	nth_finder()
head_finder	Retrieve the head of an input	head_finder()
tail_finder	Retrieve the tail of an input	tail_finder()
token_finder	Search for a matching token in an input	token_finder()
range_finder	Do no search, always returns the given range	range_finder()
regex_finder	Search for a substring matching the given regex	regex_finder()

Table 8.8. Formatters

Formatter	Description	Generators
const_formatter	Constant formatter. Always return the specified string	const_formatter()
identity_formatter	Identity formatter. Return unmodified input input	identity_formatter()
empty_formatter	Null formatter. Always return an empty string	empty_formatter()
regex_formatter		regex_formatter()

	Regex formatter. Format regex match using the specification in the format string	

3.3. Iterators

Table 8.9. Find Iterators

Iterator name	**Description**	**Iterator class**
find_iterator	Iterates through matching substrings in the input	find_iterator
split_iterator	Iterates through gaps between matching substrings in the input	split_iterator

3.4. Classification

Table 8.10. Predicates

Predicate name	**Description**	**Generator**
is_classified	Generic `ctype` mask based classification	is_classified()
is_space	Recognize spaces	is_space()
is_alnum	Recognize alphanumeric characters	is_alnum()
is_alpha	Recognize letters	is_alpha()
is_cntrl	Recognize control characters	is_cntrl()
is_digit	Recognize decimal digits	is_digit()
is_graph	Recognize graphical characters	is_graph()
is_lower	Recognize lower case characters	is_lower()
is_print	Recognize printable characters	is_print()
is_punct	Recognize punctuation characters	is_punct()
is_upper	Recognize uppercase characters	is_upper()
is_xdigit	Recognize hexadecimal digits	is_xdigit()

4. Design Topics

4.1. String Representation

As the name suggest, this library works mainly with strings. However, in the context of this library, a string is not restricted to any particular implementation (like `std::basic_string`), rather it is a concept. This allows the algorithms in this library to be reused for any string type, that satisfies the given requirements.

Definition: A string is a collection of characters accessible in sequential ordered fashion. Character is any value type with "cheap" copying and assignment.

First requirement of string−type is that it must accessible using collection traits. This facility allows to access the elements inside the string in a uniform iterator−based fashion. This requirement is actually less stringent than that of collection concept. It implements an external collection interface. This is sufficient for our library

Second requirement defines the way in which the characters are stored in the string. Algorithms in this library work with an assumption that copying a character is cheaper then allocating extra storage to cache results. This is a natural assumption for common character types. Algorithms will work even if this requirement is not satisfied, however at the cost of performance degradation.

In addition some algorithms have additional requirements on the string−type. Particularly, it is required that an algorithm can create a new string of the given type. In this case, it is required that the type satisfies the sequence (Std §23.1.1) requirements.

In the reference and also in the code, requirement on the string type is designated by the name of template argument. `CollectionT` means that the basic collection requirements must hold. `SequenceT` designates extended sequence requirements.

4.2. `iterator_range` class

An iterator_range is an encapsulation of a pair of iterators that delimit a sequence (or, a range). This concept is widely used by sequence manipulating algorithms. Although being so useful, there no direct support for it in the standard library (The closest thing is that some algorithms return a pair of iterators). Instead all STL algorithms have two distinct parameters for beginning and end of a range. This design is natural for implementation of generic algorithms, but it forbids to work with a range as a single value.

It is possible to encapsulate a range in `std::pair<>`, but `std::pair<>` is an overly generic encapsulation, so it is not best match for a range. For instance, it does not enforce that begin and end iterators be of the same type.

Naturally the range concept is heavily used also in this library. During the development of the library, it was discovered, that there is a need for a reasonable encapsulation for it, since core part of the library deals with substring searching algorithms and any such algorithm returns a range delimiting the result of the search. `std::pair<>` was deemed as unsuitable. Therefore the `iterator_range` was defined.

The intention of the `iterator_range` class is to manage a range as a single value and provide a basic interface for common operations. Its interface is similar to that of a collection. In addition to `begin()` and `end()` accessors, it has member functions for checking whether the range is empty, or to determine the size of the range. It also has a set of member typedefs that extract type information from the encapsulated iterators. As such, the interface is compatible with the collection traits requirements so it is possible to use this class as a parameter to many algorithms in this library.

iterator_range will be moved to Boost.Range library in the future releases. The internal version will be deprecated then.

4.3. Collection Traits

Collection traits provide uniform access to different types of collections . This functionality allows to write generic algorithms which work with several different kinds of collections. For this library it means, that, for instance, many algorithms work with `std::string` as well as with `char[]`. This facility implements the external collection concept.

The following collection types are supported:

- Standard containers
- Built−in arrays (like int[])
- Null terminated strings (this includes char[],wchar_t[],char*, and wchar_t*)
- std::pair<iterator,iterator>

Collection traits support a subset of the container concept (Std §23.1). This subset can be described as an input container concept, e.g. a container with immutable content. Its definition can be found in the header boost/algorithm/string/collection_traits.hpp.

In the table C denotes a container and c is an object of C.

Table 8.11. Collection Traits

Name	Standard collection equivalent	Description
value_type_of<C>::type	`C::value_type`	Type of contained values
difference_type_of<C>::type	`C::difference_type`	difference type of the collection
iterator_of<C>::type	`C::iterator`	iterator type of the collection
const_iterator_of<C>::type	`C::const_iterator`	const_iterator type of the collection
result_iterator_of<C>::type		result_iterator type of the collection. This type maps to `C::iterator` for mutable collection and `C::const_iterator` for const collection.
begin(c)	`c.begin()`	Gets the iterator pointing to the start of the collection.
end(c)	`c.end()`	Gets the iterator pointing to the end of the collection.
size(c)	`c.size()`	Gets the size of the collection.
empty(c)	`c.empty()`	Checks if the collection is empty.

The collection traits are only a temporary part of this library. They will be replaced in the future releases by Boost.Range library. Use of the internal implementation will be deprecated then.

4.4. Sequence Traits

The major difference between `std::list` and `std::vector` is not in the interfaces they provide, but rather in the inner details of the class and the way how it performs various operations. The problem is that it is not possible to infer this difference from the definitions of classes without some special mechanism. However, some algorithms can run significantly faster with the knowledge of the properties of a particular container.

Sequence traits allow one to specify additional properties of a sequence container (see Std.§32.2). These properties are then used by algorithms to select optimized handling for some operations. The sequence traits are declared in the header boost/algorithm/string/sequence_traits.hpp.

In the table C denotes a container and c is an object of C.

Table 8.12. Sequence Traits

Trait	Description
has_native_replace<C>::value	Specifies that the sequence has std::string like replace method
has_stable_iterators<C>::value	Specifies that the sequence has stable iterators. It means, that operations like `insert/erase/replace` do not invalidate iterators.
has_const_time_insert<C>::value	Specifies that the insert method of the sequence has constant time complexity.

has_const_time_erase<C>::value	Specifies that the erase method of the sequence has constant time complexity

Current implementation contains specializations for std::list<T> and std::basic_string<T> from the standard library and SGI's std::rope<T> and std::slist<T>.

4.5. Find Algorithms

Find algorithms have similar functionality to `std::search()` algorithm. They provide a different interface which is more suitable for common string operations. Instead of returning just the start of matching subsequence they return a range which is necessary when the length of the matching subsequence is not known beforehand. This feature also allows a partitioning of the input sequence into three parts: a prefix, a substring and a suffix.

Another difference is an addition of various searching methods besides find_first, including find_regex.

It the library, find algorithms are implemented in terms of Finders. Finders are used also by other facilities (replace,split). For convenience, there are also function wrappers for these finders to simplify find operations.

Currently the library contains only naive implementation of find algorithms with complexity O(n * m) where n is the size of the input sequence and m is the size of the search sequence. There are algorithms with complexity O(n), but for smaller sequence a constant overhead is rather big. For small m << n (m by magnitude smaller than n) the current implementation provides acceptable efficiency. Even the C++ standard defines the required complexity for search algorithm as O(n * m). It is possible that a future version of library will also contain algorithms with linear complexity as an option

4.6. Replace Algorithms

The implementation of replace algorithms follows the layered structure of the library. The lower layer implements generic substitution of a range in the input sequence. This layer takes a Finder object and a Formatter object as an input. These two functors define what to replace and what to replace it with. The upper layer functions are just wrapping calls to the lower layer. Finders are shared with the find and split facility.

As usual, the implementation of the lower layer is designed to work with a generic sequence while taking advantage of specific features if possible (by using Sequence traits)

4.7. Find Iterators & Split Algorithms

Find iterators are a logical extension of the find facility. Instead of searching for one match, the whole input can be iteratively searched for multiple matches. The result of the search is then used to partition the input. It depends on the algorithms which parts are returned as the result. They can be the matching parts (find_iterator) of the parts in between (split_iterator).

In addition the split algorithms like find_all() and split() can simplify the common operations. They use a find iterator to search the whole input and copy the matches they found into the supplied container.

4.8. Exception Safety

The library requires that all operations on types used as template or function arguments provide the *basic exception−safety guarantee*. In turn, all functions and algorithms in this library, except where stated otherwise, will provide the *basic exception−safety guarantee*. In other words: The library maintains its invariants and does not leak resources in the face of exceptions. Some library operations give stronger guarantees, which are documented on an individual basis.

Some functions can provide the *strong exception−safety guarantee*. That means that following statements are true:

- If an exception is thrown, there are no effects other than those of the function
- If an exception is thrown other than by the function, there are no effects

This guarantee can be provided under the condition that the operations on the types used for arguments for these functions either provide the strong exception guarantee or do not alter the global state .

In the reference, under the term *strong exception−safety guarantee*, we mean the guarantee as defined above.

For more information about the exception safety topics, follow this link

5. Concepts

5.1. Definitions

Table 8.13. Notation

`F`	A type that is a model of Finder
`Fmt`	A type that is a model of Formatter
`Iter`	Iterator Type
`f`	Object of type `F`
`fmt`	Object of type `Fmt`
`i,j`	Objects of type `Iter`

5.2. Finder Concept

Finder is a functor which searches for an arbitrary part of a container. The result of the search is given as an iterator_range delimiting the selected part.

Table 8.14. Valid Expressions

Expression	Return Type	Effects
`f(i,j)`	Convertible to `iterator_range<Iter>`	Perform the search on the interval [i,j) and returns the result of the search

Various algorithms need to perform a search in a container and a Finder is a generalization of such search operations that allows algorithms to abstract from searching. For instance, generic replace algorithms can replace any part of the input, and the Finder is used to select the desired one.

Note, that it is only required that the finder works with a particular iterator type. However, a Finder operation can be defined as a template, allowing the Finder to work with any iterator.

Examples

- Finder implemented as a class. This Finder always returns the whole input as a match. `operator()` is templated, so that the finder can be used on any iterator type.

```
struct simple_finder
{
    template<typename ForwardIteratorT>
    boost::iterator_range<ForwardIterator> operator()(
        ForwardIteratorT Begin,
        ForwardIteratorT End )
    {
        return boost::make_range( Begin, End );
    }
};
```

- Function Finder. Finder can be any function object. That is, any ordinary function with the required signature can be used as well. However, such a function can be used only for a specific iterator type.

```
boost::iterator_range<std::string> simple_finder(
    std::string::const_iterator Begin,
    std::string::const_iterator End )
{
    return boost::make_range( Begin, End );
}
```

5.3. Formatter concept

Formatters are used by replace algorithms. They are used in close combination with finders. A formatter is a functor, which takes a result from a Finder operation and transforms it in a specific way. The operation of the formatter can use additional information provided by a specific finder, for example regex_formatter() uses the match information from regex_finder() to format the result of formatter operation.

Table 8.15. Valid Expressions

Expression	Return Type	Effects
`fmt(f(i,j))`	A container type, accessible using container traits	Formats the result of the finder operation

Similarly to finders, formatters generalize format operations. When a finder is used to select a part of the input, formatter takes this selection and performs some formating on it. Algorithms can abstract from formating using a formatter.

Examples

- Formatter implemented as a class. This Formatter does not perform any formating and returns the match, repackaged. `operator()` is templated, so that the Formatter can be used on any Finder type.

```
struct simple_formatter
{
    template<typename FindResultT>
    std::string operator()( const FindResultT& Match )
    {
        std::string Temp( Match.begin(), Match.end() );
        return Temp;
    }
};
```

- Function Formatter. Similarly to Finder, Formatter can be any function object. However, as a function, it can be used only with a specific Finder type.

```
std::string simple_formatter( boost::iterator_range<std::string::const_iterator>& Match )
{
    std::string Temp( Match.begin(), Match.end() );
    return Temp;
}
```

6. Reference

6.1. Header <boost/algorithm/string/case_conv.hpp>

Defines sequence case–conversion algorithms. Algorithms convert each element in the input sequence to the desired case using provided locales.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename CollectionT> 
      OutputIteratorT 
      to_lower_copy(OutputIteratorT, const CollectionT &, 
                    const std::locale & = std::locale());
    template<typename SequenceT> 
      SequenceT to_lower_copy(const SequenceT &, 
                              const std::locale & = std::locale());
    template<typename MutableCollectionT> 
      void to_lower(MutableCollectionT &, const std::locale & = std::locale());
    template<typename OutputIteratorT, typename CollectionT> 
      OutputIteratorT 
      to_upper_copy(OutputIteratorT, const CollectionT &, 
                    const std::locale & = std::locale());
    template<typename SequenceT> 
      SequenceT to_upper_copy(const SequenceT &, 
                              const std::locale & = std::locale());
    template<typename MutableCollectionT> 
      void to_upper(MutableCollectionT &, const std::locale & = std::locale());
  }
}
```

Function to_lower_copy

boost::algorithm::to_lower_copy — Convert to lower case.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT> 
  OutputIteratorT 
  to_lower_copy(OutputIteratorT Output, const CollectionT & Input, 
                const std::locale & Loc = std::locale());
template<typename SequenceT> 
  SequenceT to_lower_copy(const SequenceT & Input, 
                          const std::locale & Loc = std::locale());
```

Description

Each element of the input sequence is converted to lower case. The result is a copy of the input converted to lower case. It is returned as a sequence or copied to the output iterator.

Parameters

Input
: An input collection

Loc
: A locale used for conversion

Output
: An output iterator to which the result will be copied

Returns

An output iterator pointing just after the last inserted character or a copy of the input

Notes

The second variant of this function provides the strong exception−safety guarantee

Function template to_lower

boost::algorithm::to_lower   Convert to lower case.

Synopsis

```
template<typename MutableCollectionT>
  void to_lower(MutableCollectionT & Input,
                const std::locale & Loc = std::locale());
```

Description

Each element of the input sequence is converted to lower case. The input sequence is modified in−place.

Parameters

Input
    A collection
Loc
    a locale used for conversion

Function to_upper_copy

boost::algorithm::to_upper_copy   Convert to upper case.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT>
  OutputIteratorT
  to_upper_copy(OutputIteratorT Output, const CollectionT & Input,
                const std::locale & Loc = std::locale());
template<typename SequenceT>
  SequenceT to_upper_copy(const SequenceT & Input,
                          const std::locale & Loc = std::locale());
```

Description

Each element of the input sequence is converted to upper case. The result is a copy of the input converted to upper case. It is returned as a sequence or copied to the output iterator

Parameters

Input
    An input collection
Loc
    A locale used for conversion
Output
    An output iterator to which the result will be copied

Returns
    An output iterator pointing just after the last inserted character or a copy of the input

Notes

The second variant of this function provides the strong exception−safety guarantee

Function template to_upper

boost::algorithm::to_upper Convert to upper case.

Synopsis

```
template<typename MutableCollectionT> 
  void to_upper(MutableCollectionT & Input, 
                const std::locale & Loc = std::locale());
```

Description

Each element of the input sequence is converted to upper case. The input sequence is modified in−place.

Parameters

Input

An input collection

Loc

a locale used for conversion

6.2. Header <boost/algorithm/string/classification.hpp>

Classification predicates are included in the library to give some more convenience when using algorithms like `trim()` and `all()` . They wrap functionality of STL classification functions (e.g. `std::isspace()`) into generic functors.

```
namespace boost {
  namespace algorithm {
    unspecified is_classified(std::ctype_base::mask, 
                              const std::locale & = std::locale());
    unspecified is_space(const std::locale & = std::locale());
    unspecified is_alnum(const std::locale & = std::locale());
    unspecified is_alpha(const std::locale & = std::locale());
    unspecified is_cntrl(const std::locale & = std::locale());
    unspecified is_digit(const std::locale & = std::locale());
    unspecified is_graph(const std::locale & = std::locale());
    unspecified is_lower(const std::locale & = std::locale());
    unspecified is_print(const std::locale & = std::locale());
    unspecified is_punct(const std::locale & = std::locale());
    unspecified is_upper(const std::locale & = std::locale());
    unspecified is_xdigit(const std::locale & = std::locale());
    template<typename ContainerT> unspecified is_any_of(const ContainerT &);
    template<typename CharT> unspecified is_from_range(CharT, CharT);
    template<typename Pred1T, typename Pred2T> 
      unspecified operator&&(const predicate_facade< Pred1T > &, 
                             const predicate_facade< Pred2T > &);
    template<typename Pred1T, typename Pred2T> 
      unspecified operator||(const predicate_facade< Pred1T > &, 
                             const predicate_facade< Pred2T > &);
    template<typename PredT> 
      unspecified operator!(const predicate_facade< PredT > &);
  }
}
```

Function is_classified

boost::algorithm::is_classified   is_classified predicate

Synopsis

```
unspecified is_classified(std::ctype_base::mask Type,
                          const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate. This predicate holds if the input is of specified `std::ctype` category.

Parameters

Loc
: A locale used for classification

Type
: A `std::ctype` category

Returns
: An instance of the `is_classified` predicate

Function is_space

boost::algorithm::is_space   is_space predicate

Synopsis

```
unspecified is_space(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::space` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_alnum

boost::algorithm::is_alnum   is_alnum predicate

Synopsis

```
unspecified is_alnum(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::alnum` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_alpha

boost::algorithm::is_alpha   is_alpha predicate

Synopsis

```
unspecified is_alpha(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::alpha` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_cntrl

boost::algorithm::is_cntrl   is_cntrl predicate

Synopsis

```
unspecified is_cntrl(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::cntrl` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_digit

boost::algorithm::is_digit — is_digit predicate

Synopsis

```
unspecified is_digit(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::digit` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_graph

boost::algorithm::is_graph — is_graph predicate

Synopsis

```
unspecified is_graph(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::graph` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_lower

boost::algorithm::is_lower — is_lower predicate

Synopsis

```
unspecified is_lower(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::lower` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of `is_classified` predicate

Function is_print

boost::algorithm::is_print   is_print predicate

Synopsis

```
unspecified is_print(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::print` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_punct

boost::algorithm::is_punct   is_punct predicate

Synopsis

```
unspecified is_punct(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::punct` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_upper

boost::algorithm::is_upper   is_upper predicate

Synopsis

```
unspecified is_upper(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::upper` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function is_xdigit

boost::algorithm::is_xdigit — is_xdigit predicate

Synopsis

```
unspecified is_xdigit(const std::locale & Loc = std::locale());
```

Description

Construct the `is_classified` predicate for the `ctype_base::xdigit` category.

Parameters

Loc
: A locale used for classification

Returns
: An instance of the `is_classified` predicate

Function template is_any_of

boost::algorithm::is_any_of — is_any_of predicate

Synopsis

```
template<typename ContainerT> unspecified is_any_of(const ContainerT & Set);
```

Description

Construct the `is_any_of` predicate. The predicate holds if the input is included in the specified set of characters.

Parameters

Set
: A set of characters to be recognized

Returns
: An instance of the `is_any_of` predicate

Function template is_from_range

boost::algorithm::is_from_range — is_from_range predicate

Synopsis

```
template<typename CharT> unspecified is_from_range(CharT From, CharT To);
```

Description

Construct the `is_from_range` predicate. The predicate holds if the input is included in the specified range. (i.e. From <= Ch <= To)

Parameters

From
: The start of the range

To
: The end of the range

Returns
: An instance of the `is_from_range` predicate

Function template operator&&

boost::algorithm::operator&& — predicate 'and' composition predicate

Synopsis

```
template<typename Pred1T, typename Pred2T>
  unspecified operator&&(const predicate_facade< Pred1T > & Pred1,
                         const predicate_facade< Pred2T > & Pred2);
```

Description

Construct the `class_and` predicate. This predicate can be used to logically combine two classification predicates. `class_and` holds, if both predicates return true.

Parameters

Pred1
: The first predicate

Pred2
: The second predicate

Returns
: An instance of the `class_and` predicate

Function template operator||

boost::algorithm::operator|| predicate 'or' composition predicate

Synopsis

```
template<typename Pred1T, typename Pred2T> 
  unspecified operator||(const predicate_facade< Pred1T > & Pred1, 
                         const predicate_facade< Pred2T > & Pred2);
```

Description

Construct the `class_or` predicate. This predicate can be used to logically combine two classification predicates. `class_or` holds, if one of the predicates return true.

Parameters

Pred1
    The first predicate
Pred2
    The second predicate

Returns
    An instance of the `class_or` predicate

Function template operator!

boost::algorithm::operator!   predicate negation operator

Synopsis

```
template<typename PredT> 
  unspecified operator!(const predicate_facade< PredT > & Pred);
```

Description

Construct the `class_not` predicate. This predicate represents a negation. `class_or` holds if of the predicates return false.

Parameters

Pred
    The predicate to be negated

Returns
    An instance of the `class_not` predicate

6.3. Header <boost/algorithm/string/collection_traits.hpp>

Defines collection_traits class and related free−standing functions. This facility is used to unify the access to different types of collections. It allows the algorithms in the library to work with STL collections, c−style array, null−terminated c−strings (and more) using the same interface.

```
namespace boost {
  namespace algorithm {
    template<typename T> struct collection_traits;
    template<typename C> struct value_type_of;
    template<typename C> struct difference_type_of;
    template<typename C> struct iterator_of;
    template<typename C> struct const_iterator_of;
    template<typename C> struct result_iterator_of;
    template<typename C> collection_traits< C >::size_type size(const C &);
    template<typename C> bool empty(const C &);
    template<typename C> collection_traits< C >::iterator begin(C &);
    template<typename C> 
      collection_traits< C >::const_iterator begin(const C &);
    template<typename C> collection_traits< C >::iterator end(C &);
    template<typename C> collection_traits< C >::const_iterator end(const C &);
  }
}
```

Struct template collection_traits

boost::algorithm::collection_traits — collection_traits class

Synopsis

```
template<typename T> 
struct collection_traits {
  // types
  typedef container_helper_type                     function_type;    // Function type.
  typedef container_helper_type::value_type         value_type;       // Value type.
  typedef container_helper_type::size_type          size_type;        // Size type.
  typedef container_helper_type::iterator           iterator;         // Iterator type.
  typedef container_helper_type::const_iterator     const_iterator;   // Const iterator type.
  typedef container_helper_type::result_iterator result_iterator;  // Result iterator type ( iterator of con
  typedef container_helper_type::difference_type difference_type;  // Difference type.
};
```

Description

Collection traits provide uniform access to different types of collections. This functionality allows to write generic algorithms which work with several different kinds of collections.

Currently following collection types are supported:

- containers with STL compatible container interface (see ContainerConcept) (i.e. `std::vector<>` , `std::list<>` , `std::string<>` ...)
- c−style array (`char` [10], `int` [15] ...)
- null−terminated c−strings (`char*` , `wchar_T*`)
- std::pair of iterators (i.e `std::pair<vector<int>::iterator` ,vector<int>::iterator>)

Collection traits provide an external collection interface operations. All are accessible using free−standing functions.

The following operations are supported:

- `size()`
- `empty()`
- `begin()`
- `end()`

Container traits have somewhat limited functionality on compilers not supporting partial template specialization and partial template ordering.

Struct template value_type_of

boost::algorithm::value_type_of — Container value_type trait.

Synopsis

```
template<typename C>
struct value_type_of {
  // types
  typedef collection_traits< C >::value_type type;
};
```

Description

Extract the type of elements contained in a container

Struct template difference_type_of

boost::algorithm::difference_type_of — Container difference trait.

Synopsis

```
template<typename C>
struct difference_type_of {
  // types
  typedef collection_traits< C >::difference_type type;
};
```

Description

Extract the container's difference type

Struct template iterator_of

boost::algorithm::iterator_of — Container iterator trait.

Synopsis

```
template<typename C>
struct iterator_of {
  // types
  typedef collection_traits< C >::iterator type;
};
```

Description

Extract the container's iterator type

Struct template const_iterator_of

boost::algorithm::const_iterator_of — Container const_iterator trait.

Synopsis

```
template<typename C>
struct const_iterator_of {
  // types
  typedef collection_traits< C >::const_iterator type;
};
```

Description

Extract the container's const_iterator type

Struct template result_iterator_of

boost::algorithm::result_iterator_of   Container result_iterator.

Synopsis

```
template<typename C>
struct result_iterator_of {
  // types
  typedef collection_traits< C >::result_iterator type;
};
```

Description

Extract the container's result_iterator type. This type maps to `C::iterator` for mutable container and `C::const_iterator` for const containers.

Function template size

boost::algorithm::size   Free–standing size() function.

Synopsis

```
template<typename C> collection_traits< C >::size_type size(const C & c);
```

Description

Get the size of the container. Uses collection_traits.

Function template empty

boost::algorithm::empty   Free–standing empty() function.

Synopsis

```
template<typename C> bool empty(const C & c);
```

Description

Check whether the container is empty. Uses container traits.

Function begin

boost::algorithm::begin   Free−standing begin() function.

Synopsis

```
template<typename C> collection_traits< C >::iterator begin(C & c);
template<typename C> collection_traits< C >::const_iterator begin(const C & c);
```

Description

Get the begin iterator of the container. Uses collection_traits.

Function end

boost::algorithm::end   Free−standing end() function.

Synopsis

```
template<typename C> collection_traits< C >::iterator end(C & c);
template<typename C> collection_traits< C >::const_iterator end(const C & c);
```

Description

Get the begin iterator of the container. Uses collection_traits.

6.4. Header <boost/algorithm/string/compare.hpp>

Defines element comparison predicates. Many algorithms in this library can take an additional argument with a predicate used to compare elements. This makes it possible, for instance, to have case insensitive versions of the algorithms.

```
namespace boost {
  namespace algorithm {
    struct is_equal;
    struct is_iequal;
  }
}
```

Struct is_equal

boost::algorithm::is_equal   is_equal functor

Synopsis

```
struct is_equal {

  // public member functions
  template<typename T1, typename T2>
    bool operator()(const T1 &, const T2 &) const;
};
```

Description

Standard STL equal_to only handle comparison between arguments of the same type. This is a less restrictive version which wraps operator ==.

`is_equal` public member functions

1.
   ```
   template<typename T1, typename T2> 
     bool operator()(const T1 & Arg1, const T2 & Arg2) const;
   ```

   Compare two operands for equality

Struct is_iequal

boost::algorithm::is_iequal — case insensitive version of is_equal

Synopsis

```
struct is_iequal {
  // construct/copy/destruct
  is_iequal(const std::locale & = std::locale());

  // public member functions
  template<typename T1, typename T2> 
    bool operator()(const T1 &, const T2 &) const;
};
```

Description

Case insensitive comparison predicate. Comparison is done using specified locales.

`is_iequal` construct/copy/destruct

1.
   ```
   is_iequal(const std::locale & Loc = std::locale());
   ```

   Parameters

   Loc

   locales used for comparison

`is_iequal` public member functions

1.
   ```
   template<typename T1, typename T2> 
     bool operator()(const T1 & Arg1, const T2 & Arg2) const;
   ```

   Compare two operands. Case is ignored.

6.5. Header <boost/algorithm/string/concept.hpp>

Defines concepts used in string_algo library

```
namespace boost {
  namespace algorithm {
    template<typename FinderT, typename IteratorT> struct FinderConcept;
    template<typename FormatterT, typename FinderT, typename IteratorT> 
      struct FormatterConcept;
  }
}
```

Struct template FinderConcept

boost::algorithm::FinderConcept — Finder concept.

Synopsis

```
template<typename FinderT, typename IteratorT>
struct FinderConcept {

  // public member functions
  void constraints() ;
};
```

Description

Defines the Finder concept. Finder is a functor which selects an arbitrary part of a string. Search is performed on the range specified by starting and ending iterators.

Result of the find operation must be convertible to iterator_range.

`FinderConcept` public member functions

1. `void constraints() ;`

Struct template FormatterConcept

boost::algorithm::FormatterConcept — Formatter concept.

Synopsis

```
template<typename FormatterT, typename FinderT, typename IteratorT>
struct FormatterConcept {

  // public member functions
  void constraints() ;
};
```

Description

Defines the Formatter concept. Formatter is a functor, which takes a result from a finder operation and transforms it in a specific way.

Result must be a container supported by container_traits, or a reference to it.

`FormatterConcept` public member functions

1. `void constraints() ;`

6.6. Header <boost/algorithm/string/constants.hpp>

```
namespace boost {
  namespace algorithm {
    enum token_compress_mode_type;
  }
}
```

Type token_compress_mode_type

boost::algorithm::token_compress_mode_type — Token compression mode.

Synopsis

```
enum token_compress_mode_type { token_compress_on, token_compress_off };
```

6.7. Header <boost/algorithm/string/erase.hpp>

Defines various erase algorithms. Each algorithm removes part(s) of the input according to a searching criteria.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename CollectionT>
      OutputIteratorT
      erase_range_copy(OutputIteratorT, const CollectionT &,
                       const iterator_range< typename const_iterator_of< CollectionT >::type > &);
    template<typename SequenceT>
      SequenceT erase_range_copy(const SequenceT &,
                                 const iterator_range< typename const_iterator_of< SequenceT >::type > &);
    template<typename SequenceT>
      void erase_range(SequenceT &,
                       const iterator_range< typename iterator_of< SequenceT >::type > &);
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T>
      OutputIteratorT
      erase_first_copy(OutputIteratorT, const Collection1T &,
                       const Collection2T &);
    template<typename SequenceT, typename CollectionT>
      SequenceT erase_first_copy(const SequenceT &, const CollectionT &);
    template<typename SequenceT, typename CollectionT>
      void erase_first(SequenceT &, const CollectionT &);
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T>
      OutputIteratorT
      ierase_first_copy(OutputIteratorT, const Collection1T &,
                        const Collection2T &,
                        const std::locale & = std::locale());
    template<typename SequenceT, typename CollectionT>
      SequenceT ierase_first_copy(const SequenceT &, const CollectionT &,
                                  const std::locale & = std::locale());
    template<typename SequenceT, typename CollectionT>
      void ierase_first(SequenceT &, const CollectionT &,
                        const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T>
      OutputIteratorT
      erase_last_copy(OutputIteratorT, const Collection1T &,
                      const Collection2T &);
    template<typename SequenceT, typename CollectionT>
      SequenceT erase_last_copy(const SequenceT &, const CollectionT &);
    template<typename SequenceT, typename CollectionT>
      void erase_last(SequenceT &, const CollectionT &);
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T>
      OutputIteratorT
      ierase_last_copy(OutputIteratorT, const Collection1T &,
                       const Collection2T &,
                       const std::locale & = std::locale());
    template<typename SequenceT, typename CollectionT>
      SequenceT ierase_last_copy(const SequenceT &, const CollectionT &,
                                 const std::locale & = std::locale());
    template<typename SequenceT, typename CollectionT>
      void ierase_last(SequenceT &, const CollectionT &,
                       const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T> 
      OutputIteratorT 
      erase_nth_copy(OutputIteratorT, const Collection1T &, 
                     const Collection2T &, unsigned int);
    template<typename SequenceT, typename CollectionT> 
      SequenceT erase_nth_copy(const SequenceT &, const CollectionT &, 
                               unsigned int);
    template<typename SequenceT, typename CollectionT> 
      void erase_nth(SequenceT &, const CollectionT &, unsigned int);
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T> 
      OutputIteratorT 
      ierase_nth_copy(OutputIteratorT, const Collection1T &, 
                      const Collection2T &, unsigned int, 
                      const std::locale & = std::locale());
    template<typename SequenceT, typename CollectionT> 
      SequenceT ierase_nth_copy(const SequenceT &, const CollectionT &, 
                                unsigned int, 
                                const std::locale & = std::locale());
    template<typename SequenceT, typename CollectionT> 
      void ierase_nth(SequenceT &, const CollectionT &, unsigned int, 
                      const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T> 
      OutputIteratorT 
      erase_all_copy(OutputIteratorT, const Collection1T &, 
                     const Collection2T &);
    template<typename SequenceT, typename CollectionT> 
      SequenceT erase_all_copy(const SequenceT &, const CollectionT &);
    template<typename SequenceT, typename CollectionT> 
      void erase_all(SequenceT &, const CollectionT &);
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T> 
      OutputIteratorT 
      ierase_all_copy(OutputIteratorT, const Collection1T &, 
                      const Collection2T &, 
                      const std::locale & = std::locale());
    template<typename SequenceT, typename CollectionT> 
      SequenceT ierase_all_copy(const SequenceT &, const CollectionT &, 
                                const std::locale & = std::locale());
    template<typename SequenceT, typename CollectionT> 
      void ierase_all(SequenceT &, const CollectionT &, 
                      const std::locale & = std::locale());
    template<typename OutputIteratorT, typename CollectionT> 
      OutputIteratorT 
      erase_head_copy(OutputIteratorT, const CollectionT &, unsigned int);
    template<typename SequenceT> 
      SequenceT erase_head_copy(const SequenceT &, unsigned int);
    template<typename SequenceT> void erase_head(SequenceT &, unsigned int);
    template<typename OutputIteratorT, typename CollectionT> 
      OutputIteratorT 
      erase_tail_copy(OutputIteratorT, const CollectionT &, unsigned int);
    template<typename SequenceT> 
      SequenceT erase_tail_copy(const SequenceT &, unsigned int);
    template<typename SequenceT> void erase_tail(SequenceT &, unsigned int);
  }
}
```

Function erase_range_copy

boost::algorithm::erase_range_copy — Erase range algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT>
  OutputIteratorT
  erase_range_copy(OutputIteratorT Output, const CollectionT & Input,
                   const iterator_range< typename const_iterator_of< CollectionT >::type > & SearchRange);
template<typename SequenceT>
  SequenceT erase_range_copy(const SequenceT & Input,
                             const iterator_range< typename const_iterator_of< SequenceT >::type > & SearchR
```

Description

Remove the given range from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Input
    An input sequence
Output
    An output iterator to which the result will be copied
SearchRange
    A range in the input to be removed

Returns
    An output iterator pointing just after the last inserted character or a modified copy of the input
Notes
    The second variant of this function provides the strong exception−safety guarantee

Function template erase_range

boost::algorithm::erase_range   Erase range algorithm.

Synopsis

```
template<typename SequenceT>
  void erase_range(SequenceT & Input,
                   const iterator_range< typename iterator_of< SequenceT >::type > & SearchRange);
```

Description

Remove the given range from the input. The input sequence is modified in−place.

Parameters

Input
    An input sequence
SearchRange
    A range in the input to be removed

Function erase_first_copy

boost::algorithm::erase_first_copy   Erase first algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T> 
  OutputIteratorT 
  erase_first_copy(OutputIteratorT Output, const Collection1T & Input, 
                   const Collection2T & Search);
template<typename SequenceT, typename CollectionT> 
  SequenceT erase_first_copy(const SequenceT & Input, 
                             const CollectionT & Search);
```

Description

Remove the first occurence of the substring from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Input
: An input string

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template erase_first

boost::algorithm::erase_first   Erase first algorithm.

Synopsis

```
template<typename SequenceT, typename CollectionT> 
  void erase_first(SequenceT & Input, const CollectionT & Search);
```

Description

Remove the first occurence of the substring from the input. The input sequence is modified in−place.

Parameters

Input
: An input string

Search
: A substring to be searched for.

Function ierase_first_copy

boost::algorithm::ierase_first_copy   Erase first algorithm (case insensitive).

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T> 
  OutputIteratorT 
  ierase_first_copy(OutputIteratorT Output, const Collection1T & Input, 
                    const Collection2T & Search, 
                    const std::locale & Loc = std::locale());
template<typename SequenceT, typename CollectionT> 
  SequenceT ierase_first_copy(const SequenceT & Input, 
                              const CollectionT & Search, 
                              const std::locale & Loc = std::locale());
```

Description

Remove the first occurence of the substring from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters

Input	An input string
Loc	A locale used for case insensitive comparison
Output	An output iterator to which the result will be copied
Search	A substring to be searched for

Returns	An output iterator pointing just after the last inserted character or a modified copy of the input
Notes	The second variant of this function provides the strong exception−safety guarantee

Function template ierase_first

boost::algorithm::ierase_first   Erase first algorithm (case insensitive).

Synopsis

```
template<typename SequenceT, typename CollectionT> 
  void ierase_first(SequenceT & Input, const CollectionT & Search, 
                    const std::locale & Loc = std::locale());
```

Description

Remove the first occurence of the substring from the input. The input sequence is modified in−place. Searching is case insensitive.

Parameters

Input	An input string
Loc	A locale used for case insensitive comparison
Search	

A substring to be searched for

Function erase_last_copy

boost::algorithm::erase_last_copy — Erase last algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T> 
  OutputIteratorT 
  erase_last_copy(OutputIteratorT Output, const Collection1T & Input, 
                  const Collection2T & Search);
template<typename SequenceT, typename CollectionT> 
  SequenceT erase_last_copy(const SequenceT & Input, 
                            const CollectionT & Search);
```

Description

Remove the last occurence of the substring from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Input
: An input string

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for.

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template erase_last

boost::algorithm::erase_last — Erase last algorithm.

Synopsis

```
template<typename SequenceT, typename CollectionT> 
  void erase_last(SequenceT & Input, const CollectionT & Search);
```

Description

Remove the last occurence of the substring from the input. The input sequence is modified in−place.

Parameters

Input
: An input string

Search
: A substring to be searched for

Function ierase_last_copy

boost::algorithm::ierase_last_copy — Erase last algorithm (case insensitive).

Synopsis

```
template<typename OutputIteratorT, typename Collection1T,
         typename Collection2T>
  OutputIteratorT
  ierase_last_copy(OutputIteratorT Output, const Collection1T & Input,
                   const Collection2T & Search,
                   const std::locale & Loc = std::locale());
template<typename SequenceT, typename CollectionT>
  SequenceT ierase_last_copy(const SequenceT & Input,
                             const CollectionT & Search,
                             const std::locale & Loc = std::locale());
```

Description

Remove the last occurence of the substring from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters

Input
: An input string

Loc
: A locale used for case insensitive comparison

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template ierase_last

boost::algorithm::ierase_last — Erase last algorithm (case insensitive).

Synopsis

```
template<typename SequenceT, typename CollectionT>
  void ierase_last(SequenceT & Input, const CollectionT & Search,
                   const std::locale & Loc = std::locale());
```

Description

Remove the last occurence of the substring from the input. The input sequence is modified in−place. Searching is case insensitive.

Parameters

Input

An input string

Loc
: A locale used for case insensitive comparison

Search
: A substring to be searched for

Function erase_nth_copy

boost::algorithm::erase_nth_copy — Erase nth algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T> 
  OutputIteratorT 
  erase_nth_copy(OutputIteratorT Output, const Collection1T & Input, 
                 const Collection2T & Search, unsigned int Nth);
template<typename SequenceT, typename CollectionT> 
  SequenceT erase_nth_copy(const SequenceT & Input, 
                           const CollectionT & Search, unsigned int Nth);
```

Description

Remove the Nth occurence of the substring in the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Input
: An input string

Nth
: An index of the match to be replaced. The index is 0−based.

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template erase_nth

boost::algorithm::erase_nth — Erase nth algorithm.

Synopsis

```
template<typename SequenceT, typename CollectionT> 
  void erase_nth(SequenceT & Input, const CollectionT & Search, 
                 unsigned int Nth);
```

Description

Remove the Nth occurence of the substring in the input. The input sequence is modified in−place.

Parameters

Input
: An input string

Nth
: An index of the match to be replaced. The index is 0−based.

Search
: A substring to be searched for.

Function ierase_nth_copy

boost::algorithm::ierase_nth_copy — Erase nth algorithm (case insensitive).

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T> 
  OutputIteratorT 
  ierase_nth_copy(OutputIteratorT Output, const Collection1T & Input, 
                  const Collection2T & Search, unsigned int Nth, 
                  const std::locale & Loc = std::locale());
template<typename SequenceT, typename CollectionT> 
  SequenceT ierase_nth_copy(const SequenceT & Input, 
                            const CollectionT & Search, unsigned int Nth, 
                            const std::locale & Loc = std::locale());
```

Description

Remove the Nth occurence of the substring in the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters

Input
: An input string

Loc
: A locale used for case insensitive comparison

Nth
: An index of the match to be replaced. The index is 0−based.

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for.

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template ierase_nth

boost::algorithm::ierase_nth   Erase nth algorithm.

Synopsis

```
template<typename SequenceT, typename CollectionT> 
  void ierase_nth(SequenceT & Input, const CollectionT & Search, 
                  unsigned int Nth, const std::locale & Loc = std::locale());
```

Description

Remove the Nth occurence of the substring in the input. The input sequence is modified in−place. Searching is case insensitive.

Parameters

Input
    An input string
Loc
    A locale used for case insensitive comparison
Nth
    An index of the match to be replaced. The index is 0−based.
Search
    A substring to be searched for.

Function erase_all_copy

boost::algorithm::erase_all_copy   Erase all algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T> 
  OutputIteratorT 
  erase_all_copy(OutputIteratorT Output, const Collection1T & Input, 
                 const Collection2T & Search);
template<typename SequenceT, typename CollectionT> 
  SequenceT erase_all_copy(const SequenceT & Input, 
                           const CollectionT & Search);
```

Description

Remove all the occurrences of the string from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Input
    An input sequence
Output
    An output iterator to which the result will be copied
Search
    A substring to be searched for.

Returns

An output iterator pointing just after the last inserted character or a modified copy of the input

Notes

The second variant of this function provides the strong exception−safety guarantee

Function template erase_all

boost::algorithm::erase_all   Erase all algorithm.

Synopsis

```
template<typename SequenceT, typename CollectionT> 
  void erase_all(SequenceT & Input, const CollectionT & Search);
```

Description

Remove all the occurrences of the string from the input. The input sequence is modified in−place.

Parameters

Input
    An input string
Search
    A substring to be searched for.

Function ierase_all_copy

boost::algorithm::ierase_all_copy   Erase all algorithm (case insensitive).

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T> 
  OutputIteratorT 
  ierase_all_copy(OutputIteratorT Output, const Collection1T & Input, 
                  const Collection2T & Search, 
                  const std::locale & Loc = std::locale());
template<typename SequenceT, typename CollectionT> 
  SequenceT ierase_all_copy(const SequenceT & Input, 
                            const CollectionT & Search, 
                            const std::locale & Loc = std::locale());
```

Description

Remove all the occurrences of the string from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters

Input
    An input string
Loc
    A locale used for case insensitive comparison
Output
    An output iterator to which the result will be copied
Search

A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template ierase_all

boost::algorithm::ierase_all   Erase all algorithm (case insensitive).

Synopsis

```
template<typename SequenceT, typename CollectionT> 
  void ierase_all(SequenceT & Input, const CollectionT & Search, 
                  const std::locale & Loc = std::locale());
```

Description

Remove all the occurrences of the string from the input. The input sequence is modified in−place. Searching is case insensitive.

Parameters

Input
: An input string

Loc
: A locale used for case insensitive comparison

Search
: A substring to be searched for.

Function erase_head_copy

boost::algorithm::erase_head_copy   Erase head algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT> 
  OutputIteratorT 
  erase_head_copy(OutputIteratorT Output, const CollectionT & Input, 
                  unsigned int N);
template<typename SequenceT> 
  SequenceT erase_head_copy(const SequenceT & Input, unsigned int N);
```

Description

Remove the head from the input. The head is a prefix of a seqence of given size. If the sequence is shorter then required, the whole string is considered to be the head. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Input
: An input string

N

Length of the head

Output

An output iterator to which the result will be copied

Returns

An output iterator pointing just after the last inserted character or a modified copy of the input

Notes

The second variant of this function provides the strong exception−safety guarantee

Function template erase_head

boost::algorithm::erase_head   Erase head algorithm.

Synopsis

```
template<typename SequenceT>
  void erase_head(SequenceT & Input, unsigned int N);
```

Description

Remove the head from the input. The head is a prefix of a seqence of given size. If the sequence is shorter then required, the whole string is considered to be the head. The input sequence is modified in−place.

Parameters

Input

An input string

N

Length of the head

Function erase_tail_copy

boost::algorithm::erase_tail_copy   Erase tail algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT>
  OutputIteratorT
  erase_tail_copy(OutputIteratorT Output, const CollectionT & Input,
                  unsigned int N);
template<typename SequenceT>
  SequenceT erase_tail_copy(const SequenceT & Input, unsigned int N);
```

Description

Remove the tail from the input. The tail is a suffix of a seqence of given size. If the sequence is shorter then required, the whole string is considered to be the tail. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Input

An input string

N

Length of the head

Output

An output iterator to which the result will be copied

Returns

An output iterator pointing just after the last inserted character or a modified copy of the input

Notes

The second variant of this function provides the strong exception−safety guarantee

Function template erase_tail

boost::algorithm::erase_tail   Erase tail algorithm.

Synopsis

```
template<typename SequenceT>
  void erase_tail(SequenceT & Input, unsigned int N);
```

Description

Remove the tail from the input. The tail is a suffix of a seqence of given size. If the sequence is shorter then required, the whole string is considered to be the tail. The input sequence is modified in−place.

Parameters

Input

An input string

N

Length of the head

6.8. Header <boost/algorithm/string/find.hpp>

Defines a set of find algorithms. The algorithms are searching for a substring of the input. The result is given as an `iterator_range` delimiting the substring.

```
namespace boost {
  namespace algorithm {
    template<typename CollectionT, typename FinderT>
      iterator_range< typename result_iterator_of< CollectionT >::type >
      find(CollectionT &, FinderT);
    template<typename Collection1T, typename Collection2T>
      iterator_range< typename result_iterator_of< Collection1T >::type >
      find_first(Collection1T &, const Collection2T &);
    template<typename Collection1T, typename Collection2T>
      iterator_range< typename result_iterator_of< Collection1T >::type >
      ifind_first(Collection1T &, const Collection2T &,
                  const std::locale & = std::locale());
    template<typename Collection1T, typename Collection2T>
      iterator_range< typename result_iterator_of< Collection1T >::type >
      find_last(Collection1T &, const Collection2T &);
    template<typename Collection1T, typename Collection2T>
      iterator_range< typename result_iterator_of< Collection1T >::type >
      ifind_last(Collection1T &, const Collection2T &,
                 const std::locale & = std::locale());
    template<typename Collection1T, typename Collection2T>
      iterator_range< typename result_iterator_of< Collection1T >::type >
      find_nth(Collection1T &, const Collection2T &, unsigned int);
    template<typename Collection1T, typename Collection2T>
      iterator_range< typename result_iterator_of< Collection1T >::type >
      ifind_nth(Collection1T &, const Collection2T &, unsigned int,
                const std::locale & = std::locale());
    template<typename CollectionT>
      iterator_range< typename result_iterator_of< CollectionT >::type >
      find_head(CollectionT &, unsigned int);
    template<typename CollectionT>
      iterator_range< typename result_iterator_of< CollectionT >::type >
      find_tail(CollectionT &, unsigned int);
    template<typename CollectionT, typename PredicateT>
      iterator_range< typename result_iterator_of< CollectionT >::type >
      find_token(CollectionT &, PredicateT,
                 token_compress_mode_type = token_compress_off);
  }
}
```

Function template find

boost::algorithm::find   Generic find algorithm.

Synopsis

```
template<typename CollectionT, typename FinderT>
  iterator_range< typename result_iterator_of< CollectionT >::type >
  find(CollectionT & Input, FinderT Finder);
```

Description

Search the input using the given finder.

Parameters

Finder
    Finder object used for searching.
Input
    A string which will be searched.

Returns
    An `iterator_range` delimiting the match. Returned iterator is either `CollectionT::iterator` or `CollectionT::const_iterator` , depending on the constness of the input parameter.

Function template find_first

boost::algorithm::find_first   Find first algorithm.

Synopsis

```
template<typename Collection1T, typename Collection2T>
  iterator_range< typename result_iterator_of< Collection1T >::type >
  find_first(Collection1T & Input, const Collection2T & Search);
```

Description

Search for the first occurence of the substring in the input.

Parameters

Input

A string which will be searched.

Search

A substring to be searched for.

Returns

An `iterator_range` delimiting the match. Returned iterator is either `CollectionT::iterator` or `CollectionT::const_iterator` , depending on the constness of the input parameter.

Notes

This function provides the strong exception–safety guarantee

Function template ifind_first

boost::algorithm::ifind_first   Find first algorithm (case insensitive).

Synopsis

```
template<typename Collection1T, typename Collection2T>
  iterator_range< typename result_iterator_of< Collection1T >::type >
  ifind_first(Collection1T & Input, const Collection2T & Search,
              const std::locale & Loc = std::locale());
```

Description

Search for the first occurence of the substring in the input. Searching is case insensitive.

Parameters

Input

A string which will be searched.

Loc

A locale used for case insensitive comparison

Search

A substring to be searched for.

Returns

An `iterator_range` delimiting the match. Returned iterator is either `Collection1T::iterator` or `Collection1T::const_iterator` , depending on the constness of the input parameter.

Notes

This function provides the strong exception–safety guarantee

Function template find_last

boost::algorithm::find_last   Find last algorithm.

Synopsis

```
template<typename Collection1T, typename Collection2T>
  iterator_range< typename result_iterator_of< Collection1T >::type >
  find_last(Collection1T & Input, const Collection2T & Search);
```

Description

Search for the last occurence of the substring in the input.

Parameters

Input
: A string which will be searched.

Search
: A substring to be searched for.

Returns
: An `iterator_range` delimiting the match. Returned iterator is either `Collection1T::iterator` or `Collection1T::const_iterator` , depending on the constness of the input parameter.

Notes
: This function provides the strong exception–safety guarantee

Function template ifind_last

boost::algorithm::ifind_last   Find last algorithm (case insensitive).

Synopsis

```
template<typename Collection1T, typename Collection2T>
  iterator_range< typename result_iterator_of< Collection1T >::type >
  ifind_last(Collection1T & Input, const Collection2T & Search,
             const std::locale & Loc = std::locale());
```

Description

Search for the last match a string in the input. Searching is case insensitive.

Parameters

Input
: A string which will be searched.

Loc
: A locale used for case insensitive comparison

Search
: A substring to be searched for.

Returns
: An `iterator_range` delimiting the match. Returned iterator is either `Collection1T::iterator` or `Collection1T::const_iterator` , depending on the constness of the input parameter.

Notes
: This function provides the strong exception–safety guarantee

Function template find_nth

boost::algorithm::find_nth   Find n–th algorithm.

Synopsis

```
template<typename Collection1T, typename Collection2T>
  iterator_range< typename result_iterator_of< Collection1T >::type >
  find_nth(Collection1T & Input, const Collection2T & Search,
           unsigned int Nth);
```

Description

Search for the n–th (zero–indexed) occurence of the substring in the input.

Parameters

Input
: A string which will be searched.

Nth
: An index (zero–indexed) of the match to be found.

Search
: A substring to be searched for.

Returns
: An `iterator_range` delimiting the match. Returned iterator is either `Collection1T::iterator` or `Collection1T::const_iterator` , depending on the constness of the input parameter.

Function template ifind_nth

boost::algorithm::ifind_nth   Find n–th algorithm (case insensitive).

Synopsis

```
template<typename Collection1T, typename Collection2T>
  iterator_range< typename result_iterator_of< Collection1T >::type >
  ifind_nth(Collection1T & Input, const Collection2T & Search,
            unsigned int Nth, const std::locale & Loc = std::locale());
```

Description

Search for the n–th (zero–indexed) occurence of the substring in the input. Searching is case insensitive.

Parameters

Input
: A string which will be searched.

Loc
: A locale used for case insensitive comparison

Nth
: An index (zero–indexed) of the match to be found.

Search
: A substring to be searched for.

Returns
: An `iterator_range` delimiting the match. Returned iterator is either `Collection1T::iterator` or `Collection1T::const_iterator` , depending on the constness of the input parameter.

Notes
: This function provides the strong exception–safety guarantee

Function template find_head

boost::algorithm::find_head   Find head algorithm.

Synopsis

```
template<typename CollectionT>
  iterator_range< typename result_iterator_of< CollectionT >::type >
  find_head(CollectionT & Input, unsigned int N);
```

Description

Get the head of the input. Head is a prefix of the string of the given size. If the input is shorter then required, whole input if considered to be the head.

Parameters

Input
: An input string

N
: Length of the head

Returns
: An `iterator_range` delimiting the match. Returned iterator is either `Collection1T::iterator` or `Collection1T::const_iterator` , depending on the constness of the input parameter.

Notes
: This function provides the strong exception–safety guarantee

Function template find_tail

boost::algorithm::find_tail Find tail algorithm.

Synopsis

```
template<typename CollectionT>
  iterator_range< typename result_iterator_of< CollectionT >::type >
  find_tail(CollectionT & Input, unsigned int N);
```

Description

Get the head of the input. Head is a suffix of the string of the given size. If the input is shorter then required, whole input if considered to be the tail.

Parameters

Input
: An input string

N
: Length of the tail

Returns
: An `iterator_range` delimiting the match. Returned iterator is either `CollectionT::iterator` or `CollectionT::const_iterator` , depending on the constness of the input parameter.

Notes
: This function provides the strong exception–safety guarantee

Function template find_token

boost::algorithm::find_token — Find token algorithm.

Synopsis

```
template<typename CollectionT, typename PredicateT>
  iterator_range< typename result_iterator_of< CollectionT >::type >
  find_token(CollectionT & Input, PredicateT Pred,
             token_compress_mode_type eCompress = token_compress_off);
```

Description

Look for a given token in the string. Token is a character that matches the given predicate. If the "token compress mode" is enabled, adjacent tokens are considered to be one match.

Parameters

Input
: A input string.

Pred
: An unary predicate to identify a token

eCompress
: Enable/Disable compressing of adjacent tokens

Returns
: An `iterator_range` delimiting the match. Returned iterator is either `CollectionT::iterator` or `CollectionT::const_iterator`, depending on the constness of the input parameter.

Notes
: This function provides the strong exception–safety guarantee

6.9. Header <boost/algorithm/string/find_format.hpp>

Defines generic replace algorithms. Each algorithm replaces part(s) of the input. The part to be replaced is looked up using a Finder object. Result of finding is then used by a Formatter object to generate the replacement.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename CollectionT, typename FinderT,
             typename FormatterT>
      OutputIteratorT
      find_format_copy(OutputIteratorT, const CollectionT &, FinderT,
                       FormatterT);
    template<typename SequenceT, typename FinderT, typename FormatterT>
      SequenceT find_format_copy(const SequenceT &, FinderT, FormatterT);
    template<typename SequenceT, typename FinderT, typename FormatterT>
      void find_format(SequenceT &, FinderT, FormatterT);
    template<typename OutputIteratorT, typename CollectionT, typename FinderT,
             typename FormatterT>
      OutputIteratorT
      find_format_all_copy(OutputIteratorT, const CollectionT &, FinderT,
                           FormatterT);
    template<typename SequenceT, typename FinderT, typename FormatterT>
      SequenceT find_format_all_copy(const SequenceT &, FinderT, FormatterT);
    template<typename SequenceT, typename FinderT, typename FormatterT>
      void find_format_all(SequenceT &, FinderT, FormatterT);
    template<typename CharT, typename RegexTraitsT, typename RegexAllocatorT>
      unspecified regex_finder(const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                               match_flag_type = match_default);
    template<typename CharT, typename TraitsT, typename AllocT>
      unspecified regex_formatter(const std::basic_string< CharT, TraitsT, AllocT > &,
                                          match_flag_type = format_default);
  }
}
```

Function find_format_copy

boost::algorithm::find_format_copy   Generic replace algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename FinderT, 
         typename FormatterT> 
  OutputIteratorT 
  find_format_copy(OutputIteratorT Output, const CollectionT & Input, 
                   FinderT Finder, FormatterT Formatter);
template<typename SequenceT, typename FinderT, typename FormatterT> 
  SequenceT find_format_copy(const SequenceT & Input, FinderT Finder, 
                             FormatterT Formatter);
```

Description

Use the Finder to search for a substring. Use the Formatter to format this substring and replace it in the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Finder	A Finder object used to search for a match to be replaced
Formatter	A Formatter object used to format a match
Input	An input sequence
Output	An output iterator to which the result will be copied

Returns	An output iterator pointing just after the last inserted character or a modified copy of the input
Notes	The second variant of this function provides the strong exception−safety guarantee

Function template find_format

boost::algorithm::find_format   Generic replace algorithm.

Synopsis

```
template<typename SequenceT, typename FinderT, typename FormatterT> 
  void find_format(SequenceT & Input, FinderT Finder, FormatterT Formatter);
```

Description

Use the Finder to search for a substring. Use the Formatter to format this substring and replace it in the input. The input is modified in−place.

Parameters

Finder
    A Finder object used to search for a match to be replaced
Formatter
    A Formatter object used to format a match
Input
    An input sequence

Function find_format_all_copy

boost::algorithm::find_format_all_copy   Generic replace all algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename FinderT,
         typename FormatterT>
  OutputIteratorT
  find_format_all_copy(OutputIteratorT Output, const CollectionT & Input,
                       FinderT Finder, FormatterT Formatter);
template<typename SequenceT, typename FinderT, typename FormatterT>
  SequenceT find_format_all_copy(const SequenceT & Input, FinderT Finder,
                                 FormatterT Formatter);
```

Description

Use the Finder to search for a substring. Use the Formatter to format this substring and replace it in the input. Repeat this for all matching substrings. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Finder
    A Finder object used to search for a match to be replaced
Formatter
    A Formatter object used to format a match
Input
    An input sequence
Output
    An output iterator to which the result will be copied

Returns
    An output iterator pointing just after the last inserted character or a modified copy of the input
Notes
    The second variant of this function provides the strong exception−safety guarantee

Function template find_format_all

boost::algorithm::find_format_all   Generic replace all algorithm.

Synopsis

```
template<typename SequenceT, typename FinderT, typename FormatterT>
  void find_format_all(SequenceT & Input, FinderT Finder,
                       FormatterT Formatter);
```

Description

Use the Finder to search for a substring. Use the Formatter to format this substring and replace it in the input. Repeat this for all matching substrings.The input is modified in−place.

Parameters

Finder
: A Finder object used to search for a match to be replaced

Formatter
: A Formatter object used to format a match

Input
: An input sequence

Function template regex_finder

boost::algorithm::regex_finder   "Regex" finder

Synopsis

```
template<typename CharT, typename RegexTraitsT, typename RegexAllocatorT>
  unspecified regex_finder(const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                           match_flag_type MatchFlags = match_default);
```

Description

Construct the `regex_finder` . Finder uses the regex engine to search for a match. Result is given in `regex_search_result` . This is an extension of the iterator_range. In addition it containes match results from the `regex_search` algorithm.

Parameters

MatchFlags
: Regex search options

Rx
: A regular expression

Returns
: An instance of the `regex_finder` object

Function template regex_formatter

boost::algorithm::regex_formatter   Regex formatter.

Synopsis

```
template<typename CharT, typename TraitsT, typename AllocT>
  unspecified regex_formatter(const std::basic_string< CharT, TraitsT, AllocT > & Format,
                              match_flag_type Flags = format_default);
```

Description

Construct the `regex_formatter` . Regex formatter uses the regex engine to format a match found by the `regex_finder` . This formatted it designed to closely cooperate with `regex_finder` .

Parameters

Flags
    Format flags
Format
    Regex format definition

Returns
    An instance of the `regex_formatter` functor

6.10. Header <boost/algorithm/string/find_iterator.hpp>

Defines find iterator classes. Find iterator repeatly applies a Finder to the specified input string to search for matches. Dereferencing the iterator yields the current match or a range between the last and the current match depending on the iterator used.

```
namespace boost {
  namespace algorithm {
    template<typename IteratorT> class find_iterator;
    template<typename IteratorT> class split_iterator;
    template<typename CollectionT, typename FinderT>
      find_iterator< typename result_iterator_of< CollectionT >::type >
      make_find_iterator(CollectionT &, FinderT);
    template<typename CollectionT, typename FinderT>
      split_iterator< typename result_iterator_of< CollectionT >::type >
      make_split_iterator(CollectionT &, FinderT);
  }
}
```

Class template find_iterator

boost::algorithm::find_iterator   find_iterator

Synopsis

```
template<typename IteratorT>
class find_iterator {
public:
  // construct/copy/destruct
  find_iterator();
  find_iterator(const find_iterator &);
  template<typename FinderT> find_iterator(IteratorT, IteratorT, FinderT);
  template<typename FinderT, typename CollectionT>
    find_iterator(CollectionT &, FinderT);

  // public member functions
  bool eof() const;

  // private member functions
  const match_type & dereference() const;
  void increment() ;
  bool equal(const find_iterator &) const;
};
```

Description

Find iterator encapsulates a Finder and allows for incremental searching in a string. Each increment moves the iterator to the next match.

Find iterator is a readable forward traversal iterator.

Dereferencing the iterator yields an iterator_range delimiting the current match.

`find_iterator` construct/copy/destruct

1. `find_iterator();`

   Construct null iterator. All null iterators are equal.

   Postconditions

   eof()==true
2. `find_iterator(const find_iterator & Other);`

   Construct a copy of the find_iterator

3.
   ```
   template<typename FinderT> 
     find_iterator(IteratorT Begin, IteratorT End, FinderT Finder);
   ```

   Construct new find_iterator for a given finder and a range.

4.
   ```
   template<typename FinderT, typename CollectionT> 
     find_iterator(CollectionT & Col, FinderT Finder);
   ```

   Construct new find_iterator for a given finder and a collection.

`find_iterator` public member functions

1. `bool eof() const;`

   Check the eof condition. Eof condition means that there is nothing more to be searched i.e. find_iterator is after the last match.

`find_iterator` private member functions

1. `const match_type & dereference() const;`
2. `void increment() ;`
3. `bool equal(const find_iterator & Other) const;`

Class template split_iterator

boost::algorithm::split_iterator   split_iterator

Synopsis

```
template<typename IteratorT> 
class split_iterator {
public:
  // construct/copy/destruct
  split_iterator();
  split_iterator(const split_iterator &);
  template<typename FinderT> split_iterator(IteratorT, IteratorT, FinderT);
  template<typename FinderT, typename CollectionT> 
    split_iterator(CollectionT &, FinderT);

  // public member functions
  bool eof() const;

  // private member functions
  const match_type & dereference() const;
  void increment() ;
  bool equal(const split_iterator &) const;
```

```
};
```

Description

Split iterator encapsulates a Finder and allows for incremental searching in a string. Unlike the find iterator, split iterator iterates through gaps between matches.

Find iterator is a readable forward traversal iterator.

Dereferencing the iterator yields an iterator_range delimiting the current match.

`split_iterator` construct/copy/destruct

1. `split_iterator();`

   Construct null iterator. All null iterators are equal.

   Postconditions

   eof()==true
2. `split_iterator(const split_iterator & Other);`

   Construct a copy of the split_iterator

3.
   ```
   template<typename FinderT>
     split_iterator(IteratorT Begin, IteratorT End, FinderT Finder);
   ```

   Construct new split_iterator for a given finder and a range.

4.
   ```
   template<typename FinderT, typename CollectionT>
     split_iterator(CollectionT & Col, FinderT Finder);
   ```

   Construct new split_iterator for a given finder and a collection.

`split_iterator` public member functions

1. `bool eof() const;`

   Check the eof condition. Eof condition means that there is nothing more to be searched i.e. find_iterator is after the last match.

`split_iterator` private member functions

1. `const match_type & dereference() const;`
2. `void increment() ;`
3. `bool equal(const split_iterator & Other) const;`

Function template make_find_iterator

boost::algorithm::make_find_iterator find iterator construction helper

Synopsis

```
template<typename CollectionT, typename FinderT>
  find_iterator< typename result_iterator_of< CollectionT >::type >
  make_find_iterator(CollectionT & Collection, FinderT Finder);
```

Description

Construct a find iterator to iterate through the specified string

Function template make_split_iterator

boost::algorithm::make_split_iterator — split iterator construction helper

Synopsis

```
template<typename CollectionT, typename FinderT> 
  split_iterator< typename result_iterator_of< CollectionT >::type > 
  make_split_iterator(CollectionT & Collection, FinderT Finder);
```

Description

Construct a split iterator to iterate through the specified collection

6.11. Header <boost/algorithm/string/finder.hpp>

Defines Finder generators. Finder object is a functor which is able to find a substring matching a specific criteria in the input. Finders are used as a pluggable components for replace, find and split facilities. This header contains generator functions for finders provided in this library.

```
namespace boost {
  namespace algorithm {
    template<typename ContainerT> unspecified first_finder(const ContainerT &);
    template<typename ContainerT, typename PredicateT> 
      unspecified first_finder(const ContainerT &, PredicateT);
    template<typename ContainerT> unspecified last_finder(const ContainerT &);
    template<typename ContainerT, typename PredicateT> 
      unspecified last_finder(const ContainerT &, PredicateT);
    template<typename ContainerT> 
      unspecified nth_finder(const ContainerT &, unsigned int);
    template<typename ContainerT, typename PredicateT> 
      unspecified nth_finder(const ContainerT &, unsigned int, PredicateT);
    unspecified head_finder(unsigned int);
    unspecified tail_finder(unsigned int);
    template<typename PredicateT> 
      unspecified token_finder(PredicateT, 
                               token_compress_mode_type = token_compress_off);
    template<typename ForwardIteratorT> 
      unspecified range_finder(ForwardIteratorT, ForwardIteratorT);
    template<typename ForwardIteratorT> 
      unspecified range_finder(iterator_range< ForwardIteratorT >);
  }
}
```

Function first_finder

boost::algorithm::first_finder — "First" finder

Synopsis

```
template<typename ContainerT> 
  unspecified first_finder(const ContainerT & Search);
template<typename ContainerT, typename PredicateT> 
  unspecified first_finder(const ContainerT & Search, PredicateT Comp);
```

Description

Construct the `first_finder` . The finder searches for the first occurrence of the string in a given input. The result is given as an `iterator_range` delimiting the match.

Parameters

Search
: A substring to be searched for.

Returns
: An instance of the `first_finder` object

Function last_finder

boost::algorithm::last_finder   "Last" finder

Synopsis

```
template<typename ContainerT> 
  unspecified last_finder(const ContainerT & Search);
template<typename ContainerT, typename PredicateT> 
  unspecified last_finder(const ContainerT & Search, PredicateT Comp);
```

Description

Construct the `last_finder` . The finder searches for the last occurrence of the string in a given input. The result is given as an `iterator_range` delimiting the match.

Parameters

Search
: A substring to be searched for.

Returns
: An instance of the `last_finder` object

Function nth_finder

boost::algorithm::nth_finder   "Nth" finder

Synopsis

```
template<typename ContainerT> 
  unspecified nth_finder(const ContainerT & Search, unsigned int Nth);
template<typename ContainerT, typename PredicateT> 
  unspecified nth_finder(const ContainerT & Search, unsigned int Nth, 
                         PredicateT Comp);
```

Description

Construct the `nth_finder` . The finder searches for the n−th (zero−indexed) occurrence of the string in a given input. The result is given as an `iterator_range` delimiting the match.

Parameters

Nth
: An index of the match to be find

Search
: A substring to be searched for.

Returns
: An instance of the `nth_finder` object

Function head_finder

boost::algorithm::head_finder — "Head" finder

Synopsis

```
unspecified head_finder(unsigned int N);
```

Description

Construct the `head_finder` . The finder returns a head of a given input. The head is a prefix of a string up to n elements in size. If an input has less then n elements, whole input is considered a head. The result is given as an `iterator_range` delimiting the match.

Parameters

N
: The size of the head

Returns
: An instance of the `head_finder` object

Function tail_finder

boost::algorithm::tail_finder — "Tail" finder

Synopsis

```
unspecified tail_finder(unsigned int N);
```

Description

Construct the `tail_finder` . The finder returns a tail of a given input. The tail is a suffix of a string up to n elements in size. If an input has less then n elements, whole input is considered a head. The result is given as an `iterator_range` delimiting the match.

Parameters

N
: The size of the head

Returns
: An instance of the `tail_finder` object

Function template token_finder

boost::algorithm::token_finder — "Token" finder

Synopsis

```
template<typename PredicateT>
  unspecified token_finder(PredicateT Pred,
                           token_compress_mode_type eCompress = token_compress_off);
```

Description

Construct the `token_finder` . The finder searches for a token specified by a predicate. It is similar to std::find_if algorithm, with an exception that it return a range of instead of a single iterator.

If "compress token mode" is enabled, adjacent matching tokens are concatenated into one match. Thus the finder can be used to search for continuous segments of characters satisfying the given predicate.

The result is given as an `iterator_range` delimiting the match.

Parameters

Pred
: An element selection predicate

eCompress
: Compress flag

Returns
: An instance of the `token_finder` object

Function range_finder

boost::algorithm::range_finder — "Range" finder

Synopsis

```
template<typename ForwardIteratorT>
  unspecified range_finder(ForwardIteratorT Begin, ForwardIteratorT End);
template<typename ForwardIteratorT>
  unspecified range_finder(iterator_range< ForwardIteratorT > Range);
```

Description

Construct the `range_finder` . The finder does not perform any operation. It simply returns the given range for any input.

Parameters

Begin
: Beginning of the range

End
: End of the range

Returns
: An instance of the `range_finger` object

6.12. Header <boost/algorithm/string/formatter.hpp>

Defines Formatter generators. Formatter is a functor which formats a string according to given parameters. A Formatter works in conjunction with a Finder. A Finder can provide additional information for a specific Formatter. An example of such a cooperation is regex_finder and regex_formatter.

Formatters are used as pluggable components for replace facilities. This header contains generator functions for the Formatters provided in this library.

```
namespace boost {
  namespace algorithm {
    template<typename CollectionT>
      unspecified const_formatter(const CollectionT &);
    template<typename CollectionT> unspecified identity_formatter();
    template<typename CollectionT>
      unspecified empty_formatter(const CollectionT &);
  }
}
```

Function template const_formatter

boost::algorithm::const_formatter — Constant formatter.

Synopsis

```
template<typename CollectionT>
  unspecified const_formatter(const CollectionT & Format);
```

Description

Construct the `const_formatter` . Const formatter always returns the same value, regardless of the parameter.

Parameters

Format
: A predefined value used as a result for formating

Returns
: An instance of the `const_formatter` object.

Function template identity_formatter

boost::algorithm::identity_formatter — Identity formatter.

Synopsis

```
template<typename CollectionT> unspecified identity_formatter();
```

Description

Construct the `identity_formatter` . Identity formatter always returns the parameter.

Returns
: An instance of the `identity_formatter` object.

Function template empty_formatter

boost::algorithm::empty_formatter — Empty formatter.

Synopsis

```
template<typename CollectionT>
  unspecified empty_formatter(const CollectionT & );
```

Description

Construct the `empty_formatter` . Empty formater always returns an empty sequence.

Returns
: An instance of the `empty_formatter` object.

6.13. Header <boost/algorithm/string/iterator_range.hpp>

Defines the `iterator_class` and related functions. `iterator_range` is a simple wrapper of the iterator pair idiom. It provides a rich subset of the Container interface.

```
namespace boost {
  namespace algorithm {
    template<typename IteratorT> class iterator_range;
    template<typename IteratorT, typename Elem, typename Traits>
      std::basic_ostream< Elem, Traits > &
      operator<<(std::basic_ostream< Elem, Traits > &,
                 const iterator_range< IteratorT > &);
    template<typename IteratorT>
      iterator_range< IteratorT > make_iterator_range(IteratorT, IteratorT);
    template<typename IteratorT>
      iterator_range< IteratorT >
      make_iterator_range(const std::pair< IteratorT, IteratorT > &);
    template<typename SeqT, typename IteratorT>
      SeqT copy_iterator_range(const iterator_range< IteratorT > &);
    template<typename SeqT, typename IteratorT, typename FuncT>
      SeqT transform_iterator_range(const iterator_range< IteratorT > &,
                                    FuncT);
  }
}
```

Class template iterator_range

boost::algorithm::iterator_range — iterator_range class

Synopsis

```
template<typename IteratorT>
class iterator_range {
public:
  // types
  typedef iterator_range< IteratorT > type;                  // this type
  typedef unspecified                 value_type;            // Encapsulated value type.
  typedef unspecified                 reference;             // Reference type.
  typedef unspecified                 difference_type;       // Difference type.
  typedef unspecified                 size_type;             // Size type.
  typedef IteratorT                   const_iterator;        // const_iterator type
  typedef IteratorT                   iterator;              // iterator type
  typedef iterator(iterator_range::*  unspecified_bool_type; // Safe bool conversion.

  // construct/copy/destruct
  iterator_range();
  iterator_range(iterator, iterator);
  iterator_range(const std::pair< IteratorT, IteratorT > &);
  iterator_range(const iterator_range &);
  template<typename OtherItT>
    iterator_range(const iterator_range< OtherItT > &);
  iterator_range& operator=(const iterator_range &);
  template<typename OtherItT>
    iterator_range& operator=(const iterator_range< OtherItT > &);

  // public member functions
  template<typename OtherItT>
    bool operator==(const iterator_range< OtherItT > &) const;
  template<typename OtherItT>
    bool operator!=(const iterator_range< OtherItT > &) const;
  IteratorT begin() const;
  IteratorT end() const;
  bool empty() const;
  difference_type size() const;
  void swap(iterator_range &) ;
  operator unspecified_bool_type() const;
};
```

Description

An `iterator_range` delimits a range in a sequence by beginning and ending iterators. An iterator_range can be passed to an algorithm which requires a sequence as an input. For example, the `toupper()` function may most frequently be used on strings, but can also be used on iterator_ranges:

```
                boost::tolower( find( s, "UPPERCASE STRING" ) );
```

Many algorithms working with sequences take a pair of iterators, delimiting a working range, as arguments. The `iterator_range` class is an encapsulation of a range identified by a pair of iterators. It provides a collection interface, so it is possible to pass an instance to an algorithm requiring a collection as an input.

`iterator_range` construct/copy/destruct

1. `iterator_range();`
2. `iterator_range(iterator Begin, iterator End);`
3. `iterator_range(const std::pair< IteratorT, IteratorT > & Range);`
4. `iterator_range(const iterator_range & Other);`

5.
   ```
   template<typename OtherItT>
     iterator_range(const iterator_range< OtherItT > & Other);
   ```

   This constructor is provided to allow conversion between const and mutable iterator instances of this class template
6. `iterator_range& operator=(const iterator_range & Other);`

7.
   ```
   template<typename OtherItT>
     iterator_range& operator=(const iterator_range< OtherItT > & Other);
   ```

`iterator_range` public member functions

1.
   ```
   template<typename OtherItT>
     bool operator==(const iterator_range< OtherItT > & Other) const;
   ```

   Compare operands for equality

2.
   ```
   template<typename OtherItT>
     bool operator!=(const iterator_range< OtherItT > & Other) const;
   ```

   Compare operands for non−equality
3. `IteratorT begin() const;`

Retrieve the begin iterator

4. `IteratorT end() const;`

   Retrieve the end iterator

5. `bool empty() const;`

   Test whether the range is empty

6. `difference_type size() const;`

   Retrieve the size of the range

7. `void swap(iterator_range & Other) ;`

   Swap two ranges

8. `operator unspecified_bool_type() const;`

Function template operator<<

boost::algorithm::operator<<   iterator_range output operator

Synopsis

```
template<typename IteratorT, typename Elem, typename Traits>
  std::basic_ostream< Elem, Traits > &
  operator<<(std::basic_ostream< Elem, Traits > & Os,
             const iterator_range< IteratorT > & Range);
```

Description

Output the range to an ostream. Elements are outputed in a sequence without separators.

Function template make_iterator_range

boost::algorithm::make_iterator_range   iterator_range construct helper

Synopsis

```
template<typename IteratorT>
  iterator_range< IteratorT >
  make_iterator_range(IteratorT Begin, IteratorT End);
```

Description

Construct an `iterator_range` from a pair of iterators

Parameters

Begin	A begin iterator
End	An end iterator

Returns
: iterator_range object

Function template make_iterator_range

boost::algorithm::make_iterator_range — iterator_range construct helper

Synopsis

```
template<typename IteratorT> 
  iterator_range< IteratorT > 
  make_iterator_range(const std::pair< IteratorT, IteratorT > & Pair);
```

Description

Construct an `iterator_range` from a `std::pair<>` containing the begin and end iterators.

Parameters

Pair
: A `std::pair<>` with begin and end iterators

Returns
: `iterator_range` object

Function template copy_iterator_range

boost::algorithm::copy_iterator_range — copy a range into a sequence

Synopsis

```
template<typename SeqT, typename IteratorT> 
  SeqT copy_iterator_range(const iterator_range< IteratorT > & Range);
```

Description

Construct a new sequence of the specified type from the elements in the given range

Parameters

Range
: An input range

Returns
: New sequence

Function template transform_iterator_range

boost::algorithm::transform_iterator_range — transform a range into a sequence

Synopsis

```
template<typename SeqT, typename IteratorT, typename FuncT> 
  SeqT transform_iterator_range(const iterator_range< IteratorT > & Range, 
                                FuncT Func);
```

Description

Create a new sequence from the elements in the range, transformed by a function

Parameters

Func
: Transformation function

Range
: An input range

Returns
: New sequence

6.14. Header <boost/algorithm/string/predicate.hpp>

Defines string–related predicates. The predicates determine whether a substring is contained in the input string under various conditions: a string starts with the substring, ends with the substring, simply contains the substring or if both strings are equal. Additionaly the algorithm `all()` checks all elements of a container to satisfy a condition.

All predicates provide the strong exception guarantee.

```
namespace boost {
  namespace algorithm {
    template<typename Collection1T, typename Collection2T, 
             typename PredicateT> 
      bool starts_with(const Collection1T &, const Collection2T &, PredicateT);
    template<typename Collection1T, typename Collection2T> 
      bool starts_with(const Collection1T &, const Collection2T &);
    template<typename Collection1T, typename Collection2T> 
      bool istarts_with(const Collection1T &, const Collection2T &, 
                        const std::locale & = std::locale());
    template<typename Collection1T, typename Collection2T, 
             typename PredicateT> 
      bool ends_with(const Collection1T &, const Collection2T &, PredicateT);
    template<typename Collection1T, typename Collection2T> 
      bool ends_with(const Collection1T &, const Collection2T &);
    template<typename Collection1T, typename Collection2T> 
      bool iends_with(const Collection1T &, const Collection2T &, 
                      const std::locale & = std::locale());
    template<typename Collection1T, typename Collection2T, 
             typename PredicateT> 
      bool contains(const Collection1T &, const Collection2T &, PredicateT);
    template<typename Collection1T, typename Collection2T> 
      bool contains(const Collection1T &, const Collection2T &);
    template<typename Collection1T, typename Collection2T> 
      bool icontains(const Collection1T &, const Collection2T &, 
                     const std::locale & = std::locale());
    template<typename Collection1T, typename Collection2T, 
             typename PredicateT> 
      bool equals(const Collection1T &, const Collection2T &, PredicateT);
    template<typename Collection1T, typename Collection2T> 
      bool equals(const Collection1T &, const Collection2T &);
    template<typename Collection1T, typename Collection2T> 
      bool iequals(const Collection1T &, const Collection2T &, 
                   const std::locale & = std::locale());
    template<typename CollectionT, typename PredicateT> 
      bool all(const CollectionT &, PredicateT);
  }
}
```

Function starts_with

boost::algorithm::starts_with — 'Starts with' predicate

Synopsis

```
template<typename Collection1T, typename Collection2T, typename PredicateT>
  bool starts_with(const Collection1T & Input, const Collection2T & Test,
                   PredicateT Comp);
template<typename Collection1T, typename Collection2T>
  bool starts_with(const Collection1T & Input, const Collection2T & Test);
```

Description

This predicate holds when the test collection is a prefix of the Input. In other words, if the input starts with the test. When the optional predicate is specified, it is used for character−wise comparison.

Parameters

Comp
: An element comparison predicate

Input
: An input sequence

Test
: A test sequence

Returns
: The result of the test

Notes
: This function provides the strong exception−safety guarantee

Function template istarts_with

boost::algorithm::istarts_with — 'Starts with' predicate (case insensitive)

Synopsis

```
template<typename Collection1T, typename Collection2T>
  bool istarts_with(const Collection1T & Input, const Collection2T & Test,
                    const std::locale & Loc = std::locale());
```

Description

This predicate holds when the test container is a prefix of the Input. In other words, if the input starts with the test. Elements are compared case insensitively.

Parameters

Input
: An input sequence

Loc
: A locale used for case insensitive comparison

Test
: A test sequence

Returns

The result of the test

Notes

This function provides the strong exception−safety guarantee

Function ends_with

boost::algorithm::ends_with 'Ends with' predicate

Synopsis

```
template<typename Collection1T, typename Collection2T, typename PredicateT>
  bool ends_with(const Collection1T & Input, const Collection2T & Test,
                 PredicateT Comp);
template<typename Collection1T, typename Collection2T>
  bool ends_with(const Collection1T & Input, const Collection2T & Test);
```

Description

This predicate holds when the test container is a suffix of the Input. In other words, if the input ends with the test. When the optional predicate is specified, it is used for character−wise comparison.

Parameters

Comp

An element comparison predicate

Input

An input sequence

Test

A test sequence

Returns

The result of the test

Notes

This function provides the strong exception−safety guarantee

Function template iends_with

boost::algorithm::iends_with 'Ends with' predicate (case insensitive)

Synopsis

```
template<typename Collection1T, typename Collection2T>
  bool iends_with(const Collection1T & Input, const Collection2T & Test,
                  const std::locale & Loc = std::locale());
```

Description

This predicate holds when the test container is a suffix of the Input. In other words, if the input ends with the test. Elements are compared case insensitively.

Parameters

Input

An input sequence

Loc

A locale used for case insensitive comparison

Test

A test sequence

Returns

The result of the test

Notes

This function provides the strong exception−safety guarantee

Function contains

boost::algorithm::contains 'Contains' predicate

Synopsis

```
template<typename Collection1T, typename Collection2T, typename PredicateT> 
  bool contains(const Collection1T & Input, const Collection2T & Test, 
                PredicateT Comp);
template<typename Collection1T, typename Collection2T> 
  bool contains(const Collection1T & Input, const Collection2T & Test);
```

Description

This predicate holds when the test container is contained in the Input. When the optional predicate is specified, it is used for character−wise comparison.

Parameters

Comp

An element comparison predicate

Input

An input sequence

Test

A test sequence

Returns

The result of the test

Notes

This function provides the strong exception−safety guarantee

Function template icontains

boost::algorithm::icontains 'Contains' predicate (case insensitive)

Synopsis

```
template<typename Collection1T, typename Collection2T> 
  bool icontains(const Collection1T & Input, const Collection2T & Test, 
                 const std::locale & Loc = std::locale());
```

Description

This predicate holds when the test container is contained in the Input. Elements are compared case insensitively.

Parameters

Input
: An input sequence

Loc
: A locale used for case insensitive comparison

Test
: A test sequence

Returns
: The result of the test

Notes
: This function provides the strong exception−safety guarantee

Function equals

boost::algorithm::equals 'Equals' predicate

Synopsis

```
template<typename Collection1T, typename Collection2T, typename PredicateT>
  bool equals(const Collection1T & Input, const Collection2T & Test,
              PredicateT Comp);
template<typename Collection1T, typename Collection2T>
  bool equals(const Collection1T & Input, const Collection2T & Test);
```

Description

This predicate holds when the test container is equal to the input container i.e. all elements in both containers are same. When the optional predicate is specified, it is used for character−wise comparison.

Parameters

Comp
: An element comparison predicate

Input
: An input sequence

Test
: A test sequence

Returns
: The result of the test

Notes
: This is a two−way version of `std::equal` algorithm

  This function provides the strong exception−safety guarantee

Function template iequals

boost::algorithm::iequals 'Equals' predicate (casa insensitive)

Synopsis

```
template<typename Collection1T, typename Collection2T>
  bool iequals(const Collection1T & Input, const Collection2T & Test,
               const std::locale & Loc = std::locale());
```

Description

This predicate holds when the test container is equal to the input container i.e. all elements in both containers are same. Elements are compared case insensitively.

Parameters

Input
: An input sequence

Loc
: A locale used for case insensitive comparison

Test
: A test sequence

Returns
: The result of the test

Notes
: This is a two−way version of `std::equal` algorithm

: This function provides the strong exception−safety guarantee

Function template all

boost::algorithm::all 'All' predicate

Synopsis

```
template<typename CollectionT, typename PredicateT>
  bool all(const CollectionT & Input, PredicateT Pred);
```

Description

This predicate holds it all its elements satisfy a given condition, represented by the predicate.

Parameters

Input
: An input sequence

Pred
: A predicate

Returns
: The result of the test

Notes
: This function provides the strong exception−safety guarantee

6.15. Header <boost/algorithm/string/regex.hpp>

Defines regex variants of the algorithms.

```
namespace boost {
  namespace algorithm {
    template<typename CollectionT, typename CharT, typename RegexTraitsT,
             typename RegexAllocatorT>
      iterator_range< typename result_iterator_of< CollectionT >::type >
      find_regex(CollectionT &,
                 const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                 match_flag_type = match_default);
    template<typename OutputIteratorT, typename CollectionT, typename CharT,
             typename RegexTraitsT, typename RegexAllocatorT,
             typename FormatStringTraitsT, typename FormatStringAllocatorT>
      OutputIteratorT
      replace_regex_copy(OutputIteratorT, const CollectionT &,
                         const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                         const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT > &,
                         match_flag_type = match_default|format_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename RegexAllocatorT, typename FormatStringTraitsT,
             typename FormatStringAllocatorT>
      SequenceT replace_regex_copy(const SequenceT &,
                                   const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                                   const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocato
                                   match_flag_type = match_default|format_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename RegexAllocatorT, typename FormatStringTraitsT,
             typename FormatStringAllocatorT>
      void replace_regex(SequenceT &,
                         const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                         const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT > &,
                         match_flag_type = match_default|format_default);
    template<typename OutputIteratorT, typename CollectionT, typename CharT,
             typename RegexTraitsT, typename RegexAllocatorT,
             typename FormatStringTraitsT, typename FormatStringAllocatorT>
      OutputIteratorT
      replace_all_regex_copy(OutputIteratorT, const CollectionT &,
                             const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                             const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT > &
                             match_flag_type = match_default|format_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename RegexAllocatorT, typename FormatStringTraitsT,
             typename FormatStringAllocatorT>
      SequenceT replace_all_regex_copy(const SequenceT &,
                                       const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                                       const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllo
                                       match_flag_type = match_default|format_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename RegexAllocatorT, typename FormatStringTraitsT,
             typename FormatStringAllocatorT>
      void replace_all_regex(SequenceT &,
                             const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                             const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT > &
                             match_flag_type = match_default|format_default);
    template<typename OutputIteratorT, typename CollectionT, typename CharT,
             typename RegexTraitsT, typename RegexAllocatorT>
      OutputIteratorT
      erase_regex_copy(OutputIteratorT, const CollectionT &,
                       const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                       match_flag_type = match_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename RegexAllocatorT>
      SequenceT erase_regex_copy(const SequenceT &,
                                 const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                                 match_flag_type = match_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT,
             typename RegexAllocatorT>
      void erase_regex(SequenceT &,
                       const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                       match_flag_type = match_default);
    template<typename OutputIteratorT, typename CollectionT, typename CharT,
             typename RegexTraitsT, typename RegexAllocatorT>
      OutputIteratorT
      erase_all_regex_copy(OutputIteratorT, const CollectionT &,
                           const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                           match_flag_type = match_default);
```

```
    template<typename SequenceT, typename CharT, typename RegexTraitsT, 
             typename RegexAllocatorT> 
      SequenceT erase_all_regex_copy(const SequenceT &, 
                                     const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &, 
                                     match_flag_type = match_default);
    template<typename SequenceT, typename CharT, typename RegexTraitsT, 
             typename RegexAllocatorT> 
      void erase_all_regex(SequenceT &, 
                           const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &, 
                           match_flag_type = match_default);
    template<typename SequenceSequenceT, typename CollectionT, typename CharT, 
             typename RegexTraitsT, typename RegexAllocatorT> 
      SequenceSequenceT & 
      find_all_regex(SequenceSequenceT &, const CollectionT &, 
                     const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &, 
                     match_flag_type = match_default);
    template<typename SequenceSequenceT, typename CollectionT, typename CharT, 
             typename RegexTraitsT, typename RegexAllocatorT> 
      SequenceSequenceT & 
      split_regex(SequenceSequenceT &, const CollectionT &, 
                  const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &, 
                  match_flag_type = match_default);
  }
}
```

Function template find_regex

boost::algorithm::find_regex — Find regex algorithm.

Synopsis

```
template<typename CollectionT, typename CharT, typename RegexTraitsT, 
         typename RegexAllocatorT> 
  iterator_range< typename result_iterator_of< CollectionT >::type > 
  find_regex(CollectionT & Input, 
             const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx, 
             match_flag_type Flags = match_default);
```

Description

Search for a substring matching the given regex in the input.

Parameters

Flags
: Regex options

Input
: A container which will be searched.

Rx
: A regular expression

Returns
: An `iterator_range` delimiting the match. Returned iterator is either `InputContainerT::iterator` or `InputContainerT::const_iterator` , depending on the constness of the input parameter.

Notes
: This function provides the strong exception–safety guarantee

Function replace_regex_copy

boost::algorithm::replace_regex_copy — Replace regex algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename CharT,
         typename RegexTraitsT, typename RegexAllocatorT,
         typename FormatStringTraitsT, typename FormatStringAllocatorT>
  OutputIteratorT
  replace_regex_copy(OutputIteratorT Output, const CollectionT & Input,
                     const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                     const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT > & Format,
                     match_flag_type Flags = match_default|format_default);
template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename RegexAllocatorT, typename FormatStringTraitsT,
         typename FormatStringAllocatorT>
  SequenceT replace_regex_copy(const SequenceT & Input,
                               const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                               const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT >
                               match_flag_type Flags = match_default|format_default);
```

Description

Search for a substring matching given regex and format it with the specified format. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Flags
    Regex options
Format
    Regex format definition
Input
    An input string
Output
    An output iterator to which the result will be copied
Rx
    A regular expression

Returns
    An output iterator pointing just after the last inserted character or a modified copy of the input
Notes
    The second variant of this function provides the strong exception–safety guarantee

Function template replace_regex

boost::algorithm::replace_regex — Replace regex algorithm.

Synopsis

```
template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename RegexAllocatorT, typename FormatStringTraitsT,
         typename FormatStringAllocatorT>
  void replace_regex(SequenceT & Input,
                     const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                     const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT > & Format,
                     match_flag_type Flags = match_default|format_default);
```

Description

Search for a substring matching given regex and format it with the specified format. The input string is modified in−place.

Parameters

Flags
: Regex options

Format
: Regex format definition

Input
: An input string

Rx
: A regular expression

Function replace_all_regex_copy

boost::algorithm::replace_all_regex_copy   Replace all regex algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename CharT,
         typename RegexTraitsT, typename RegexAllocatorT,
         typename FormatStringTraitsT, typename FormatStringAllocatorT>
  OutputIteratorT
  replace_all_regex_copy(OutputIteratorT Output, const CollectionT & Input,
                         const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                         const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT > & For
                         match_flag_type Flags = match_default|format_default);
template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename RegexAllocatorT, typename FormatStringTraitsT,
         typename FormatStringAllocatorT>
  SequenceT replace_all_regex_copy(const SequenceT & Input,
                                   const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                                   const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocato
                                   match_flag_type Flags = match_default|format_default);
```

Description

Format all substrings, matching given regex, with the specified format. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Flags
: Regex options

Format
: Regex format definition

Input
: An input string

Output
: An output iterator to which the result will be copied

Rx
: A regular expression

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes

The second variant of this function provides the strong exception−safety guarantee

Function template replace_all_regex

boost::algorithm::replace_all_regex — Replace all regex algorithm.

Synopsis

```
template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename RegexAllocatorT, typename FormatStringTraitsT,
         typename FormatStringAllocatorT>
  void replace_all_regex(SequenceT & Input,
                         const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                         const std::basic_string< CharT, FormatStringTraitsT, FormatStringAllocatorT > & For
                         match_flag_type Flags = match_default|format_default);
```

Description

Format all substrings, matching given regex, with the specified format. The input string is modified in−place.

Parameters

Flags
: Regex options

Format
: Regex format definition

Input
: An input string

Rx
: A regular expression

Function erase_regex_copy

boost::algorithm::erase_regex_copy — Erase regex algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename CharT,
         typename RegexTraitsT, typename RegexAllocatorT>
  OutputIteratorT
  erase_regex_copy(OutputIteratorT Output, const CollectionT & Input,
                   const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                   match_flag_type Flags = match_default);
template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename RegexAllocatorT>
  SequenceT erase_regex_copy(const SequenceT & Input,
                             const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                             match_flag_type Flags = match_default);
```

Description

Remove a substring matching given regex from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Flags
: Regex options

Input
: An input string

Output
: An output iterator to which the result will be copied

Rx
: A regular expression

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template erase_regex

boost::algorithm::erase_regex   Erase regex algorithm.

Synopsis

```
template<typename SequenceT, typename CharT, typename RegexTraitsT, 
         typename RegexAllocatorT> 
  void erase_regex(SequenceT & Input, 
                   const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx, 
                   match_flag_type Flags = match_default);
```

Description

Remove a substring matching given regex from the input. The input string is modified in−place.

Parameters

Flags
: Regex options

Input
: An input string

Rx
: A regular expression

Function erase_all_regex_copy

boost::algorithm::erase_all_regex_copy   Erase all regex algorithm.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename CharT, 
         typename RegexTraitsT, typename RegexAllocatorT> 
  OutputIteratorT 
  erase_all_regex_copy(OutputIteratorT Output, const CollectionT & Input, 
                       const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx, 
                       match_flag_type Flags = match_default);
template<typename SequenceT, typename CharT, typename RegexTraitsT, 
         typename RegexAllocatorT> 
  SequenceT erase_all_regex_copy(const SequenceT & Input, 
                                 const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx, 
                                 match_flag_type Flags = match_default);
```

Description

Erase all substrings, matching given regex, from the input. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Flags
: Regex options

Input
: An input string

Output
: An output iterator to which the result will be copied

Rx
: A regular expression

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template erase_all_regex

boost::algorithm::erase_all_regex — Erase all regex algorithm.

Synopsis

```
template<typename SequenceT, typename CharT, typename RegexTraitsT,
         typename RegexAllocatorT>
  void erase_all_regex(SequenceT & Input,
                       const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                       match_flag_type Flags = match_default);
```

Description

Erase all substrings, matching given regex, from the input. The input string is modified in−place.

Parameters

Flags
: Regex options

Input
: An input string

Rx
: A regular expression

Function template find_all_regex

boost::algorithm::find_all_regex — Find all regex algorithm.

Synopsis

```
template<typename SequenceSequenceT, typename CollectionT, typename CharT,
         typename RegexTraitsT, typename RegexAllocatorT>
  SequenceSequenceT &
  find_all_regex(SequenceSequenceT & Result, const CollectionT & Input,
                 const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                 match_flag_type Flags = match_default);
```

Description

This algorithm finds all substrings matching the give regex in the input.

Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Parameters

Flags
: Regex options

Input
: A container which will be searched.

Result
: A container that can hold copies of references to the substrings.

Rx
: A regular expression

Returns
: A reference to the result

Notes
: Prior content of the result will be overwritten.

  This function provides the strong exception−safety guarantee

Function template split_regex

boost::algorithm::split_regex Split regex algorithm.

Synopsis

```
template<typename SequenceSequenceT, typename CollectionT, typename CharT,
         typename RegexTraitsT, typename RegexAllocatorT>
  SequenceSequenceT &
  split_regex(SequenceSequenceT & Result, const CollectionT & Input,
              const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
              match_flag_type Flags = match_default);
```

Description

Tokenize expression. This function is equivalent to C strtok. Input sequence is split into tokens, separated by separators. Separator is an every match of the given regex. Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Parameters

Flags
: Regex options

Input
: A container which will be searched.

Result
: A container that can hold copies of references to the substrings.

Rx
: A regular expression

Returns
: A reference to the result

Notes
: Prior content of the result will be overwritten.

  This function provides the strong exception–safety guarantee

6.16. Header <boost/algorithm/string/regex_find_format.hpp>

Defines the `regex_finder` and `regex_formatter` generators. These two functors are designed to work together. `regex_formatter` uses additional information about a match contained in the regex_finder search result.

```
namespace boost {
  namespace algorithm {
    template<typename CharT, typename RegexTraitsT, typename RegexAllocatorT>
      unspecified regex_finder(const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > &,
                               match_flag_type = match_default);
    template<typename CharT, typename TraitsT, typename AllocT>
      unspecified regex_formatter(const std::basic_string< CharT, TraitsT, AllocT > &,
                                  match_flag_type = format_default);
  }
}
```

Function template regex_finder

boost::algorithm::regex_finder "Regex" finder

Synopsis

```
template<typename CharT, typename RegexTraitsT, typename RegexAllocatorT>
  unspecified regex_finder(const reg_expression< CharT, RegexTraitsT, RegexAllocatorT > & Rx,
                           match_flag_type MatchFlags = match_default);
```

Description

Construct the `regex_finder` . Finder uses the regex engine to search for a match. Result is given in `regex_search_result` . This is an extension of the iterator_range. In addition it containes match results from the `regex_search` algorithm.

Parameters

MatchFlags
: Regex search options

Rx
: A regular expression

Returns
: An instance of the `regex_finder` object

Function template regex_formatter

boost::algorithm::regex_formatter — Regex formatter.

Synopsis

```
template<typename CharT, typename TraitsT, typename AllocT>
  unspecified regex_formatter(const std::basic_string< CharT, TraitsT, AllocT > & Format,
                              match_flag_type Flags = format_default);
```

Description

Construct the `regex_formatter` . Regex formatter uses the regex engine to format a match found by the `regex_finder` . This formatted it designed to closely cooperate with `regex_finder` .

Parameters

Flags	Format flags
Format	Regex format definition

Returns
: An instance of the `regex_formatter` functor

6.17. Header <boost/algorithm/string/replace.hpp>

Defines various replace algorithms. Each algorithm replaces part(s) of the input according to set of searching and replace criteria.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T>
      OutputIteratorT
      replace_range_copy(OutputIteratorT, const Collection1T &,
                         const iterator_range< typename const_iterator_of< Collection1T >::type > &,
                         const Collection2T &);
    template<typename SequenceT, typename CollectionT>
      SequenceT replace_range_copy(const SequenceT &,
                                   const iterator_range< typename const_iterator_of< SequenceT >::type > &,
                                   const CollectionT &);
    template<typename SequenceT, typename CollectionT>
      void replace_range(SequenceT &,
                         const iterator_range< typename iterator_of< SequenceT >::type > &,
                         const CollectionT &);
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T, typename Collection3T>
      OutputIteratorT
      replace_first_copy(OutputIteratorT, const Collection1T &,
                         const Collection2T &, const Collection3T &);
    template<typename SequenceT, typename Collection1T, typename Collection2T>
      SequenceT replace_first_copy(const SequenceT &, const Collection1T &,
                                   const Collection2T &);
    template<typename SequenceT, typename Collection1T, typename Collection2T>
      void replace_first(SequenceT &, const Collection1T &,
                         const Collection2T &);
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T, typename Collection3T>
      OutputIteratorT
      ireplace_first_copy(OutputIteratorT, const Collection1T &,
                          const Collection2T &, const Collection3T &,
                          const std::locale & = std::locale());
    template<typename SequenceT, typename Collection2T, typename Collection1T> 
      SequenceT ireplace_first_copy(const SequenceT &, const Collection2T &, 
                                    const Collection1T &, 
                                    const std::locale & = std::locale());
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      void ireplace_first(SequenceT &, const Collection1T &, 
                          const Collection2T &, 
                          const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T, typename Collection3T> 
      OutputIteratorT 
      replace_last_copy(OutputIteratorT, const Collection1T &, 
                        const Collection2T &, const Collection3T &);
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      SequenceT replace_last_copy(const SequenceT &, const Collection1T &, 
                                  const Collection2T &);
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      void replace_last(SequenceT &, const Collection1T &, 
                        const Collection2T &);
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T, typename Collection3T> 
      OutputIteratorT 
      ireplace_last_copy(OutputIteratorT, const Collection1T &, 
                         const Collection2T &, const Collection3T &, 
                         const std::locale & = std::locale());
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      SequenceT ireplace_last_copy(const SequenceT &, const Collection1T &, 
                                   const Collection2T &, 
                                   const std::locale & = std::locale());
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      void ireplace_last(SequenceT &, const Collection1T &, 
                         const Collection2T &, 
                         const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T, typename Collection3T> 
      OutputIteratorT 
      replace_nth_copy(OutputIteratorT, const Collection1T &, 
                       const Collection2T &, unsigned int, 
                       const Collection3T &);
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      SequenceT replace_nth_copy(const SequenceT &, const Collection1T &, 
                                 unsigned int, const Collection2T &);
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      void replace_nth(SequenceT &, const Collection1T &, unsigned int, 
                       const Collection2T &);
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T, typename Collection3T> 
      OutputIteratorT 
      ireplace_nth_copy(OutputIteratorT, const Collection1T &, 
                        const Collection2T &, unsigned int, 
                        const Collection3T &, 
                        const std::locale & = std::locale());
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      SequenceT ireplace_nth_copy(const SequenceT &, const Collection1T &, 
                                  unsigned int, const Collection2T &, 
                                  const std::locale & = std::locale());
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      void ireplace_nth(SequenceT &, const Collection1T &, unsigned int, 
                        const Collection2T &, 
                        const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Collection1T, 
             typename Collection2T, typename Collection3T> 
      OutputIteratorT 
      replace_all_copy(OutputIteratorT, const Collection1T &, 
                       const Collection2T &, const Collection3T &);
    template<typename SequenceT, typename Collection1T, typename Collection2T> 
      SequenceT replace_all_copy(const SequenceT &, const Collection1T &, 
                                  const Collection2T &);
    template<typename SequenceT, typename Collection1T, typename Collection2T>
      void replace_all(SequenceT &, const Collection1T &,
                       const Collection2T &);
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T, typename Collection3T>
      OutputIteratorT
      ireplace_all_copy(OutputIteratorT, const Collection1T &,
                        const Collection2T &, const Collection3T &,
                        const std::locale & = std::locale());
    template<typename SequenceT, typename Collection1T, typename Collection2T>
      SequenceT ireplace_all_copy(const SequenceT &, const Collection1T &,
                                  const Collection2T &,
                                  const std::locale & = std::locale());
    template<typename SequenceT, typename Collection1T, typename Collection2T>
      void ireplace_all(SequenceT &, const Collection1T &,
                        const Collection2T &,
                        const std::locale & = std::locale());
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T>
      OutputIteratorT
      replace_head_copy(OutputIteratorT, const Collection1T &, unsigned int,
                        const Collection2T &);
    template<typename SequenceT, typename CollectionT>
      SequenceT replace_head_copy(const SequenceT &, unsigned int,
                                  const CollectionT &);
    template<typename SequenceT, typename CollectionT>
      void replace_head(SequenceT &, unsigned int, const CollectionT &);
    template<typename OutputIteratorT, typename Collection1T,
             typename Collection2T>
      OutputIteratorT
      replace_tail_copy(OutputIteratorT, const Collection1T &, unsigned int,
                        const Collection2T &);
    template<typename SequenceT, typename CollectionT>
      SequenceT replace_tail_copy(const SequenceT &, unsigned int,
                                  const CollectionT &);
    template<typename SequenceT, typename CollectionT>
      void replace_tail(SequenceT &, unsigned int, const CollectionT &);
  }
}
```

Function replace_range_copy

boost::algorithm::replace_range_copy — Replace range algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T,
         typename Collection2T>
  OutputIteratorT
  replace_range_copy(OutputIteratorT Output, const Collection1T & Input,
                     const iterator_range< typename const_iterator_of< Collection1T >::type > & SearchRange,
                     const Collection2T & Format);
template<typename SequenceT, typename CollectionT>
  SequenceT replace_range_copy(const SequenceT & Input,
                               const iterator_range< typename const_iterator_of< SequenceT >::type > & Searc
                               const CollectionT & Format);
```

Description

Replace the given range in the input string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Format
: A substitute string

Input
: An input string

Output
: An output iterator to which the result will be copied

SearchRange
: A range in the input to be substituted

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception–safety guarantee

Function template replace_range

boost::algorithm::replace_range   Replace range algorithm.

Synopsis

```
template<typename SequenceT, typename CollectionT>
  void replace_range(SequenceT & Input,
                     const iterator_range< typename iterator_of< SequenceT >::type > & SearchRange,
                     const CollectionT & Format);
```

Description

Replace the given range in the input string. The input sequence is modified in–place.

Parameters

Format
: A substitute string

Input
: An input string

SearchRange
: A range in the input to be substituted

Function replace_first_copy

boost::algorithm::replace_first_copy   Replace first algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T,
         typename Collection2T, typename Collection3T>
  OutputIteratorT
  replace_first_copy(OutputIteratorT Output, const Collection1T & Input,
                     const Collection2T & Search,
                     const Collection3T & Format);
template<typename SequenceT, typename Collection1T, typename Collection2T>
  SequenceT replace_first_copy(const SequenceT & Input,
                               const Collection1T & Search,
                               const Collection2T & Format);
```

Description

Replace the first match of the search substring in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Format
: A substitute string

Input
: An input string

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template replace_first

boost::algorithm::replace_first   Replace first algorithm.

Synopsis

```
template<typename SequenceT, typename Collection1T, typename Collection2T>
  void replace_first(SequenceT & Input, const Collection1T & Search,
                     const Collection2T & Format);
```

Description

replace the first match of the search substring in the input with the format string. The input sequence is modified in−place.

Parameters

Format
: A substitute string

Input
: An input string

Search
: A substring to be searched for

Function ireplace_first_copy

boost::algorithm::ireplace_first_copy   Replace first algorithm (case insensitive).

Synopsis

```
template<typename OutputIteratorT, typename Collection1T,
         typename Collection2T, typename Collection3T>
  OutputIteratorT
  ireplace_first_copy(OutputIteratorT Output, const Collection1T & Input,
                      const Collection2T & Search,
                      const Collection3T & Format,
                      const std::locale & Loc = std::locale());
template<typename SequenceT, typename Collection2T, typename Collection1T>
  SequenceT ireplace_first_copy(const SequenceT & Input,
                                const Collection2T & Search,
                                const Collection1T & Format,
                                const std::locale & Loc = std::locale());
```

Description

Replace the first match of the search substring in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters

Format	A substitute string
Input	An input string
Loc	A locale used for case insensitive comparison
Output	An output iterator to which the result will be copied
Search	A substring to be searched for

Returns	An output iterator pointing just after the last inserted character or a modified copy of the input
Notes	The second variant of this function provides the strong exception−safety guarantee

Function template ireplace_first

boost::algorithm::ireplace_first — Replace first algorithm (case insensitive).

Synopsis

```
template<typename SequenceT, typename Collection1T, typename Collection2T>
  void ireplace_first(SequenceT & Input, const Collection1T & Search,
                      const Collection2T & Format,
                      const std::locale & Loc = std::locale());
```

Description

Replace the first match of the search substring in the input with the format string. Input sequence is modified in−place. Searching is case insensitive.

Parameters

Format	A substitute string
Input	An input string
Loc	A locale used for case insensitive comparison
Search	A substring to be searched for

Function replace_last_copy

boost::algorithm::replace_last_copy — Replace last algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T,
         typename Collection2T, typename Collection3T>
  OutputIteratorT
  replace_last_copy(OutputIteratorT Output, const Collection1T & Input,
                    const Collection2T & Search, const Collection3T & Format);
template<typename SequenceT, typename Collection1T, typename Collection2T>
  SequenceT replace_last_copy(const SequenceT & Input,
                              const Collection1T & Search,
                              const Collection2T & Format);
```

Description

Replace the last match of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Format
    A substitute string
Input
    An input string
Output
    An output iterator to which the result will be copied
Search
    A substring to be searched for

Returns
    An output iterator pointing just after the last inserted character or a modified copy of the input
Notes
    The second variant of this function provides the strong exception−safety guarantee

Function template replace_last

boost::algorithm::replace_last — Replace last algorithm.

Synopsis

```
template<typename SequenceT, typename Collection1T, typename Collection2T>
  void replace_last(SequenceT & Input, const Collection1T & Search,
                    const Collection2T & Format);
```

Description

Replace the last match of the search string in the input with the format string. Input sequence is modified in−place.

Parameters

Format
    A substitute string
Input

An input string

Search
: A substring to be searched for

Function ireplace_last_copy

boost::algorithm::ireplace_last_copy — Replace last algorithm (case insensitive).

Synopsis

```
template<typename OutputIteratorT, typename Collection1T,
         typename Collection2T, typename Collection3T>
  OutputIteratorT
  ireplace_last_copy(OutputIteratorT Output, const Collection1T & Input,
                     const Collection2T & Search,
                     const Collection3T & Format,
                     const std::locale & Loc = std::locale());
template<typename SequenceT, typename Collection1T, typename Collection2T>
  SequenceT ireplace_last_copy(const SequenceT & Input,
                               const Collection1T & Search,
                               const Collection2T & Format,
                               const std::locale & Loc = std::locale());
```

Description

Replace the last match of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters

Format
: A substitute string

Input
: An input string

Loc
: A locale used for case insensitive comparison

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template ireplace_last

boost::algorithm::ireplace_last — Replace last algorithm (case insensitive).

Synopsis

```
template<typename SequenceT, typename Collection1T, typename Collection2T>
  void ireplace_last(SequenceT & Input, const Collection1T & Search,
                     const Collection2T & Format,
                     const std::locale & Loc = std::locale());
```

Description

Replace the last match of the search string in the input with the format string.The input sequence is modified in−place. Searching is case insensitive.

Parameters

Format
: A substitute string

Input
: An input string

Loc
: A locale used for case insensitive comparison

Search
: A substring to be searched for

Returns
: A reference to the modified input

Function replace_nth_copy

boost::algorithm::replace_nth_copy   Replace nth algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T, typename Collection3T> 
  OutputIteratorT 
  replace_nth_copy(OutputIteratorT Output, const Collection1T & Input, 
                   const Collection2T & Search, unsigned int Nth, 
                   const Collection3T & Format);
template<typename SequenceT, typename Collection1T, typename Collection2T> 
  SequenceT replace_nth_copy(const SequenceT & Input, 
                             const Collection1T & Search, unsigned int Nth, 
                             const Collection2T & Format);
```

Description

Replace an Nth (zero−indexed) match of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Format
: A substitute string

Input
: An input string

Nth
: An index of the match to be replaced. The index is 0−based.

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes

The second variant of this function provides the strong exception−safety guarantee

Function template replace_nth

boost::algorithm::replace_nth   Replace nth algorithm.

Synopsis

```
template<typename SequenceT, typename Collection1T, typename Collection2T>
  void replace_nth(SequenceT & Input, const Collection1T & Search,
                   unsigned int Nth, const Collection2T & Format);
```

Description

Replace an Nth (zero−indexed) match of the search string in the input with the format string. Input sequence is modified in−place.

Parameters

Format
    A substitute string
Input
    An input string
Nth
    An index of the match to be replaced. The index is 0−based.
Search
    A substring to be searched for

Function ireplace_nth_copy

boost::algorithm::ireplace_nth_copy   Replace nth algorithm (case insensitive).

Synopsis

```
template<typename OutputIteratorT, typename Collection1T,
         typename Collection2T, typename Collection3T>
  OutputIteratorT
  ireplace_nth_copy(OutputIteratorT Output, const Collection1T & Input,
                    const Collection2T & Search, unsigned int Nth,
                    const Collection3T & Format,
                    const std::locale & Loc = std::locale());
template<typename SequenceT, typename Collection1T, typename Collection2T>
  SequenceT ireplace_nth_copy(const SequenceT & Input,
                              const Collection1T & Search, unsigned int Nth,
                              const Collection2T & Format,
                              const std::locale & Loc = std::locale());
```

Description

Replace an Nth (zero−indexed) match of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters

Format
    A substitute string

Input
: An input string

Loc
: A locale used for case insensitive comparison

Nth
: An index of the match to be replaced. The index is 0−based.

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template ireplace_nth

boost::algorithm::ireplace_nth — Replace nth algorithm (case insensitive).

Synopsis

```
template<typename SequenceT, typename Collection1T, typename Collection2T> 
  void ireplace_nth(SequenceT & Input, const Collection1T & Search, 
                    unsigned int Nth, const Collection2T & Format, 
                    const std::locale & Loc = std::locale());
```

Description

Replace an Nth (zero−indexed) match of the search string in the input with the format string. Input sequence is modified in−place. Searching is case insensitive.

Parameters

Format
: A substitute string

Input
: An input string

Loc
: A locale used for case insensitive comparison

Nth
: An index of the match to be replaced. The index is 0−based.

Search
: A substring to be searched for

Function replace_all_copy

boost::algorithm::replace_all_copy — Replace all algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T, typename Collection3T> 
  OutputIteratorT 
  replace_all_copy(OutputIteratorT Output, const Collection1T & Input, 
                   const Collection2T & Search, const Collection3T & Format);
```

```
template<typename SequenceT, typename Collection1T, typename Collection2T>
  SequenceT replace_all_copy(const SequenceT & Input,
                             const Collection1T & Search,
                             const Collection2T & Format);
```

Description

Replace all occurrences of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Format
: A substitute string

Input
: An input string

Output
: An output iterator to which the result will be copied

Search
: A substring to be searched for

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template replace_all

boost::algorithm::replace_all   Replace all algorithm.

Synopsis

```
template<typename SequenceT, typename Collection1T, typename Collection2T>
  void replace_all(SequenceT & Input, const Collection1T & Search,
                   const Collection2T & Format);
```

Description

Replace all occurrences of the search string in the input with the format string. The input sequence is modified in−place.

Parameters

Format
: A substitute string

Input
: An input string

Search
: A substring to be searched for

Returns
: A reference to the modified input

Function ireplace_all_copy

boost::algorithm::ireplace_all_copy   Replace all algorithm (case insensitive).

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T, typename Collection3T> 
  OutputIteratorT 
  ireplace_all_copy(OutputIteratorT Output, const Collection1T & Input, 
                    const Collection2T & Search, const Collection3T & Format, 
                    const std::locale & Loc = std::locale());
template<typename SequenceT, typename Collection1T, typename Collection2T> 
  SequenceT ireplace_all_copy(const SequenceT & Input, 
                              const Collection1T & Search, 
                              const Collection2T & Format, 
                              const std::locale & Loc = std::locale());
```

Description

Replace all occurrences of the search string in the input with the format string. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator. Searching is case insensitive.

Parameters

Format
    A substitute string
Input
    An input string
Loc
    A locale used for case insensitive comparison
Output
    An output iterator to which the result will be copied
Search
    A substring to be searched for

Returns
    An output iterator pointing just after the last inserted character or a modified copy of the input
Notes
    The second variant of this function provides the strong exception−safety guarantee

Function template ireplace_all

boost::algorithm::ireplace_all   Replace all algorithm (case insensitive).

Synopsis

```
template<typename SequenceT, typename Collection1T, typename Collection2T> 
  void ireplace_all(SequenceT & Input, const Collection1T & Search, 
                    const Collection2T & Format, 
                    const std::locale & Loc = std::locale());
```

Description

Replace all occurrences of the search string in the input with the format string.The input sequence is modified in−place. Searching is case insensitive.

Parameters

Format
: A substitute string

Input
: An input string

Loc
: A locale used for case insensitive comparison

Search
: A substring to be searched for

Function replace_head_copy

boost::algorithm::replace_head_copy — Replace head algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T, 
         typename Collection2T> 
  OutputIteratorT 
  replace_head_copy(OutputIteratorT Output, const Collection1T & Input, 
                    unsigned int N, const Collection2T & Format);
template<typename SequenceT, typename CollectionT> 
  SequenceT replace_head_copy(const SequenceT & Input, unsigned int N, 
                              const CollectionT & Format);
```

Description

Replace the head of the input with the given format string. The head is a prefix of a string of given size. If the sequence is shorter then required, whole string if considered to be the head. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Format
: A substitute string

Input
: An input string

N
: Length of the head

Output
: An output iterator to which the result will be copied

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template replace_head

boost::algorithm::replace_head — Replace head algorithm.

Synopsis

```
template<typename SequenceT, typename CollectionT>
  void replace_head(SequenceT & Input, unsigned int N,
                    const CollectionT & Format);
```

Description

Replace the head of the input with the given format string. The head is a prefix of a string of given size. If the sequence is shorter then required, the whole string is considered to be the head. The input sequence is modified in−place.

Parameters

Format
: A substitute string

Input
: An input string

N
: Length of the head

Function replace_tail_copy

boost::algorithm::replace_tail_copy — Replace tail algorithm.

Synopsis

```
template<typename OutputIteratorT, typename Collection1T,
         typename Collection2T>
  OutputIteratorT
  replace_tail_copy(OutputIteratorT Output, const Collection1T & Input,
                    unsigned int N, const Collection2T & Format);
template<typename SequenceT, typename CollectionT>
  SequenceT replace_tail_copy(const SequenceT & Input, unsigned int N,
                              const CollectionT & Format);
```

Description

Replace the tail of the input with the given format string. The tail is a suffix of a string of given size. If the sequence is shorter then required, whole string is considered to be the tail. The result is a modified copy of the input. It is returned as a sequence or copied to the output iterator.

Parameters

Format
: A substitute string

Input
: An input string

N
: Length of the tail

Output
: An output iterator to which the result will be copied

Returns
: An output iterator pointing just after the last inserted character or a modified copy of the input

Notes
: The second variant of this function provides the strong exception−safety guarantee

Function template replace_tail

boost::algorithm::replace_tail   Replace tail algorithm.

Synopsis

```
template<typename SequenceT, typename CollectionT> 
  void replace_tail(SequenceT & Input, unsigned int N, 
                    const CollectionT & Format);
```

Description

Replace the tail of the input with the given format sequence. The tail is a suffix of a string of given size. If the sequence is shorter then required, the whole string is considered to be the tail. The input sequence is modified in–place.

Parameters

Format	A substitute string
Input	An input string
N	Length of the tail

6.18. Header <boost/algorithm/string/sequence_traits.hpp>

Traits defined in this header are used by various algorithms to achieve better performance for specific containers. Traits provide fail–safe defaults. If a container supports some of these features, it is possible to specialize the specific trait for this container. For lacking compilers, it is possible of define an override for a specific tester function.

Due to a language restriction, it is not currently possible to define specializations for stl containers without including the corresponding header. To decrease the overhead needed by this inclusion, user can selectively include a specialization header for a specific container. They are located in boost/algorithm/string/stl directory. Alternatively she can include boost/algorithm/string/std_collection_traits.hpp header which contains specializations for all stl containers.

```
namespace boost {
  namespace algorithm {
    template<typename T> class has_native_replace;
    template<typename T> class has_stable_iterators;
    template<typename T> class has_const_time_insert;
    template<typename T> class has_const_time_erase;
  }
}
```

Class template has_native_replace

boost::algorithm::has_native_replace   Native replace trait.

Synopsis

```
template<typename T> 
class has_native_replace {
public:
  // types
  typedef mpl::bool_< value > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = false) ;
};
```

Description

This trait specifies that the sequence has `std::string` like replace method

`has_native_replace` public member functions

1. `BOOST_STATIC_CONSTANT(bool , value  = false) ;`

Class template has_stable_iterators

boost::algorithm::has_stable_iterators   Stable iterators trait.

Synopsis

```
template<typename T> 
class has_stable_iterators {
public:
  // types
  typedef mpl::bool_< value > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = false) ;
};
```

Description

This trait specifies that the sequence has stable iterators. It means that operations like insert/erase/replace do not invalidate iterators.

`has_stable_iterators` public member functions

1. `BOOST_STATIC_CONSTANT(bool , value  = false) ;`

Class template has_const_time_insert

boost::algorithm::has_const_time_insert   Const time insert trait.

Synopsis

```
template<typename T> 
class has_const_time_insert {
public:
  // types
  typedef mpl::bool_< value > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = false) ;
};
```

Description

This trait specifies that the sequence's insert method has constant time complexity.

`has_const_time_insert` public member functions

1. `BOOST_STATIC_CONSTANT(bool , value  = false) ;`

Class template has_const_time_erase

boost::algorithm::has_const_time_erase — Const time erase trait.

Synopsis

```
template<typename T>
class has_const_time_erase {
public:
  // types
  typedef mpl::bool_< value > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = false) ;
};
```

Description

This trait specifies that the sequence's erase method has constant time complexity.

`has_const_time_erase` public member functions

1. `BOOST_STATIC_CONSTANT(bool , value  = false) ;`

6.19. Header <boost/algorithm/string/split.hpp>

Defines basic split algorithms. Split algorithms can be used to divide a string into several parts according to given criteria.

Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

```
namespace boost {
  namespace algorithm {
    template<typename SequenceSequenceT, typename Collection1T,
             typename Collection2T>
      SequenceSequenceT &
      find_all(SequenceSequenceT &, Collection1T &, const Collection2T &);
    template<typename SequenceSequenceT, typename Collection1T,
             typename Collection2T>
      SequenceSequenceT &
      ifind_all(SequenceSequenceT &, Collection1T &, const Collection2T &,
                const std::locale & = std::locale());
    template<typename SequenceSequenceT, typename CollectionT,
             typename PredicateT>
      SequenceSequenceT &
      split(SequenceSequenceT &, CollectionT &, PredicateT,
            token_compress_mode_type = token_compress_off);
  }
}
```

Function template find_all

boost::algorithm::find_all   Find all algorithm.

Synopsis

```
template<typename SequenceSequenceT, typename Collection1T,
         typename Collection2T>
  SequenceSequenceT &
  find_all(SequenceSequenceT & Result, Collection1T & Input,
           const Collection2T & Search);
```

Description

This algorithm finds all occurrences of the search string in the input.

Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Parameters

Input
: A container which will be searched.

Result
: A container that can hold copies of references to the substrings

Search
: A substring to be searched for.

Returns
: A reference the result

Notes
: Prior content of the result will be overwritten.

: This function provides the strong exception–safety guarantee

Function template ifind_all

boost::algorithm::ifind_all   Find all algorithm (case insensitive).

Synopsis

```
template<typename SequenceSequenceT, typename Collection1T,
         typename Collection2T>
  SequenceSequenceT &
  ifind_all(SequenceSequenceT & Result, Collection1T & Input,
            const Collection2T & Search,
            const std::locale & Loc = std::locale());
```

Description

This algorithm finds all occurrences of the search string in the input. Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Searching is case insensitive.

Parameters

Input
: A container which will be searched.

Loc
: A locale used for case insensitive comparison

Result
: A container that can hold copies of references to the substrings

Search
: A substring to be searched for.

Returns
: A reference the result

Notes
: Prior content of the result will be overwritten.

  This function provides the strong exception−safety guarantee

Function template split

boost::algorithm::split   Split algorithm.

Synopsis

```
template<typename SequenceSequenceT, typename CollectionT,
         typename PredicateT>
  SequenceSequenceT &
  split(SequenceSequenceT & Result, CollectionT & Input, PredicateT Pred,
        token_compress_mode_type eCompress = token_compress_off);
```

Description

Tokenize expression. This function is equivalent to C strtok. Input sequence is split into tokens, separated by separators. Separators are given by means of the predicate.

Each part is copied and added as a new element to the output container. Thus the result container must be able to hold copies of the matches (in a compatible structure like std::string) or a reference to it (e.g. using the iterator range class). Examples of such a container are `std::vector<std::string>` or `std::list<boost::iterator_range<std::string::iterator>>`

Parameters

Input
: A container which will be searched.

Pred
: A predicate to identify separators. This predicate is supposed to return true if a given element is a separator.

Result

A container that can hold copies of references to the substrings

eCompress
: If eCompress argument is set to token_compress_on, adjacent separators are merged together. Otherwise, every two separators delimit a token.

Returns
: A reference the result

Notes
: Prior content of the result will be overwritten.

  This function provides the strong exception−safety guarantee

6.20. Header <boost/algorithm/string/std_containers_traits.hpp>

This file includes sequence traits for stl containers.

6.21. Header <boost/algorithm/string.hpp>

Cumulative include for string_algo library

6.22. Header <boost/algorithm/string_regex.hpp>

Cumulative include for string_algo library. In addtion to string.hpp contains also regex−related stuff.

6.23. Header <boost/algorithm/string/trim.hpp>

Defines trim algorithms. Trim algorithms are used to remove trailing and leading spaces from a sequence (string). Space is recognized using given locales.

Parametric (`_if`) variants use a predicate (functor) to select which characters are to be trimmed.. Functions take a selection predicate as a parameter, which is used to determine whether a character is a space. Common predicates are provided in classification.hpp header.

```
namespace boost {
  namespace algorithm {
    template<typename OutputIteratorT, typename CollectionT,
             typename PredicateT>
      OutputIteratorT
      trim_left_copy_if(OutputIteratorT, const CollectionT &, PredicateT);
    template<typename SequenceT, typename PredicateT>
      SequenceT trim_left_copy_if(const SequenceT &, PredicateT);
    template<typename SequenceT>
      SequenceT trim_left_copy(const SequenceT &,
                               const std::locale & = std::locale());
    template<typename SequenceT, typename PredicateT>
      void trim_left_if(SequenceT &, PredicateT);
    template<typename SequenceT>
      void trim_left(SequenceT &, const std::locale & = std::locale());
    template<typename OutputIteratorT, typename CollectionT,
             typename PredicateT>
      OutputIteratorT
      trim_right_copy_if(OutputIteratorT, const CollectionT &, PredicateT);
    template<typename SequenceT, typename PredicateT>
      SequenceT trim_right_copy_if(const SequenceT &, PredicateT);
    template<typename SequenceT>
      SequenceT trim_right_copy(const SequenceT &,
                                const std::locale & = std::locale());
    template<typename SequenceT, typename PredicateT>
      void trim_right_if(SequenceT &, PredicateT);
    template<typename SequenceT>
      void trim_right(SequenceT &, const std::locale & = std::locale());
    template<typename OutputIteratorT, typename CollectionT, 
             typename PredicateT> 
      OutputIteratorT 
      trim_copy_if(OutputIteratorT, const CollectionT &, PredicateT);
    template<typename SequenceT, typename PredicateT> 
      SequenceT trim_copy_if(const SequenceT &, PredicateT);
    template<typename SequenceT> 
      SequenceT trim_copy(const SequenceT &, 
                          const std::locale & = std::locale());
    template<typename SequenceT, typename PredicateT> 
      void trim_if(SequenceT &, PredicateT);
    template<typename SequenceT> 
      void trim(SequenceT &, const std::locale & = std::locale());
  }
}
```

Function trim_left_copy_if

boost::algorithm::trim_left_copy_if Left trim – parametric.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename PredicateT> 
  OutputIteratorT 
  trim_left_copy_if(OutputIteratorT Output, const CollectionT & Input, 
                    PredicateT IsSpace);
template<typename SequenceT, typename PredicateT> 
  SequenceT trim_left_copy_if(const SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all leading spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The result is a trimmed copy of the input. It is returned as a sequence or copied to the output iterator

Parameters

Input
: An input collection

IsSpace
: An unary predicate identifying spaces

Output
: An output iterator to which the result will be copied

Returns
: An output iterator pointing just after the last inserted character or a copy of the input

Notes
: The second variant of this function provides the strong exception–safety guarantee

Function template trim_left_copy

boost::algorithm::trim_left_copy Left trim – parametric.

Synopsis

```
template<typename SequenceT> 
  SequenceT trim_left_copy(const SequenceT & Input, 
                           const std::locale & Loc = std::locale());
```

Description

Remove all leading spaces from the input. The result is a trimmed copy of the input.

Parameters

Input
: An input sequence

Loc
: a locale used for 'space' classification

Returns
: A trimmed copy of the input

Notes
: This function provides the strong exception−safety guarantee

Function template trim_left_if

boost::algorithm::trim_left_if   Left trim.

Synopsis

```
template<typename SequenceT, typename PredicateT> 
  void trim_left_if(SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all leading spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The input sequence is modified in−place.

Parameters

Input
: An input sequence

IsSpace
: An unary predicate identifying spaces

Function template trim_left

boost::algorithm::trim_left   Left trim.

Synopsis

```
template<typename SequenceT> 
  void trim_left(SequenceT & Input, const std::locale & Loc = std::locale());
```

Description

Remove all leading spaces from the input. The Input sequence is modified in−place.

Parameters

Input
: An input sequence

Loc
: A locale used for 'space' classification

Function trim_right_copy_if

boost::algorithm::trim_right_copy_if   Right trim – parametric.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename PredicateT>
  OutputIteratorT
  trim_right_copy_if(OutputIteratorT Output, const CollectionT & Input,
                     PredicateT IsSpace);
template<typename SequenceT, typename PredicateT>
  SequenceT trim_right_copy_if(const SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all trailing spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The result is a trimmed copy of the input. It is returned as a sequence or copied to the output iterator

Parameters

Input
: An input collection

IsSpace
: An unary predicate identifying spaces

Output
: An output iterator to which the result will be copied

Returns
: An output iterator pointing just after the last inserted character or a copy of the input

Notes
: The second variant of this function provides the strong exception–safety guarantee

Function template trim_right_copy

boost::algorithm::trim_right_copy   Right trim.

Synopsis

```
template<typename SequenceT>
  SequenceT trim_right_copy(const SequenceT & Input,
                            const std::locale & Loc = std::locale());
```

Description

Remove all trailing spaces from the input. The result is a trimmed copy of the input

Parameters

Input
: An input sequence

Loc
: A locale used for 'space' classification

Returns
: A trimmed copy of the input

Notes
: This function provides the strong exception−safety guarantee

Function template trim_right_if

boost::algorithm::trim_right_if Right trim − parametric.

Synopsis

```
template<typename SequenceT, typename PredicateT>
  void trim_right_if(SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all trailing spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The input sequence is modified in−place.

Parameters

Input
: An input sequence

IsSpace
: An unary predicate identifying spaces

Function template trim_right

boost::algorithm::trim_right Right trim.

Synopsis

```
template<typename SequenceT>
  void trim_right(SequenceT & Input, const std::locale & Loc = std::locale());
```

Description

Remove all trailing spaces from the input. The input sequence is modified in−place.

Parameters

Input
: An input sequence

Loc
: A locale used for 'space' classification

Function trim_copy_if

boost::algorithm::trim_copy_if Trim − parametric.

Synopsis

```
template<typename OutputIteratorT, typename CollectionT, typename PredicateT>
  OutputIteratorT
  trim_copy_if(OutputIteratorT Output, const CollectionT & Input,
               PredicateT IsSpace);
template<typename SequenceT, typename PredicateT>
  SequenceT trim_copy_if(const SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all trailing and leading spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The result is a trimmed copy of the input. It is returned as a sequence or copied to the output iterator

Parameters

Input	An input collection
IsSpace	An unary predicate identifying spaces
Output	An output iterator to which the result will be copied

Returns
An output iterator pointing just after the last inserted character or a copy of the input
Notes
The second variant of this function provides the strong exception−safety guarantee

Function template trim_copy

boost::algorithm::trim_copy   Trim.

Synopsis

```
template<typename SequenceT>
  SequenceT trim_copy(const SequenceT & Input,
                      const std::locale & Loc = std::locale());
```

Description

Remove all leading and trailing spaces from the input. The result is a trimmed copy of the input

Parameters

Input	An input sequence
Loc	A locale used for 'space' classification

Returns
A trimmed copy of the input
Notes
This function provides the strong exception−safety guarantee

Function template trim_if

boost::algorithm::trim_if   Trim.

Synopsis

```
template<typename SequenceT, typename PredicateT>
  void trim_if(SequenceT & Input, PredicateT IsSpace);
```

Description

Remove all leading and trailing spaces from the input. The supplied predicate is used to determine which characters are considered spaces. The input sequence is modified in−place.

Parameters

Input
    An input sequence
IsSpace
    An unary predicate identifying spaces

Function template trim

boost::algorithm::trim   Trim.

Synopsis

```
template<typename SequenceT>
  void trim(SequenceT & Input, const std::locale & Loc = std::locale());
```

Description

Remove all leading and trailing spaces from the input. The input sequence is modified in−place.

Parameters

Input
    An input sequence
Loc
    A locale used for 'space' classification

7. Rationale

7.1. Locales

Locales have a very close relation to string processing. They contain information about the character sets and are used, for example, to change the case of characters and to classify the characters.

C++ allows to work with multiple different instances of locales at once. If an algorithm manipulates some data in a way that requires the usage of locales, there must be a way to specify them. However, one instance of locales is sufficient for most of the applications, and for a user it could be very tedious to specify which locales to use at every place where it is needed.

Fortunately, the C++ standard allows to specify the *global* locales (using static member function `std:locale::global()`). When instantiating an `std::locale` class without explicit information, the instance will be initialized with the *global* locale. This implies, that if an algorithm needs a locale, it should have an `std::locale` parameter defaulting to `std::locale()`. If a user needs to specify locales explicitly, she can do so. Otherwise the *global* locales are used.

7.2. Regular Expressions

Regular expressions are an essential part of text processing. For this reason, the library also provides regex variants of some algorithms. The library does not attempt to replace Boost.Regex; it merely wraps its functionality in a new interface. As a part of this library, regex algorithms integrate smoothly with other components, which brings additional value.

8. Environment

8.1. Build

The whole library is provided in headers. Regex variants of some algorithms, however, are dependent on the Boost.Regex library. All such algorithms are separated in boost/algorithm/string_regex.hpp. If this header is used, the application must be linked with the Boost.Regex library.

8.2. Examples

Examples showing the basic usage of the library can be found in the libs/algorithm/string/example directory. There is a separate file for the each part of the library. Please follow the boost build guidelines to build examples using the bjam. To successfully build regex examples the Boost.Regex library is required.

8.3. Tests

A full set of test cases for the library is located in the libs/algorithm/string/test directory. The test cases can be executed using the boost build system. For the tests of regular expression variants of algorithms, the Boost.Regex library is required.

8.4. Portability

The library has been successfully compiled and tested with the following compilers:

- Microsoft Visual C++ 7.0
- Microsoft Visual C++ 7.1
- GCC 3.2
- GCC 3.3.1

See Boost regression tables for additional info for a particular compiler.

There are known limitation on platforms not supporting partial template specialization. Library depends on correctly implemented `std::iterator_traits` class. If a standard library provided with compiler is broken, the String Algorithm Library cannot function properly. Usually it implies that primitive pointer iterators are not working with the library functions.

9. Credits

9.1. Acknowledgments

The author would like to thank everybody who gave suggestions and comments. Especially valuable were the contributions of Thorsten Ottosen, Jeff Garland and the other boost members who participated in the review process, namely David Abrahams, Daniel Frey, Beman Dawes, John Maddock, David B.Held, Pavel Vozenilek and many other.

Additional thanks go to Stefan Slapeta and Toon Knapen, who have been very resourceful in solving various portability issues.

Chapter 9. Boost.Threads

William E. Kempf

1. Overview

1.1. Introduction

Boost.Threads allows C++ programs to execute as multiple, asynchronous, independent threads−of−execution. Each thread has its own machine state including program instruction counter and registers. Programs which execute as multiple threads are called multithreaded programs to distinguish them from traditional single−threaded programs. The glossary gives a more complete description of the multithreading execution environment.

Multithreading provides several advantages:

- Programs which would otherwise block waiting for some external event can continue to respond if the blocking operation is placed in a separate thread. Multithreading is usually an absolute requirement for these programs.
- Well−designed multithreaded programs may execute faster than single−threaded programs, particularly on multiprocessor hardware. Note, however, that poorly−designed multithreaded programs are often slower than single−threaded programs.
- Some program designs may be easier to formulate using a multithreaded approach. After all, the real world is asynchronous!

1.2. Dangers

1.2.1. General considerations

Beyond the errors which can occur in single−threaded programs, multithreaded programs are subject to additional errors:

- Race conditions
- Deadlock (sometimes called "deadly embrace")
- Priority failures (priority inversion, infinite overtaking, starvation, etc.)

Every multithreaded program must be designed carefully to avoid these errors. These aren't rare or exotic failures − they are virtually guaranteed to occur unless multithreaded code is designed to avoid them. Priority failures are somewhat less common, but are nonetheless serious.

The **Boost.Threads** design attempts to minimize these errors, but they will still occur unless the programmer proactively designs to avoid them.

Note

Please also see Section 9, Implementation Notes for additional, implementation−specific considerations.

1.2.2. Testing and debugging considerations

Multithreaded programs are non−deterministic. In other words, the same program with the same input data may follow different execution paths each time it is invoked. That can make testing and debugging a nightmare:

- Failures are often not repeatable.
- Probe effect causes debuggers to produce very different results from non−debug uses.
- Debuggers require special support to show thread state.
- Tests on a single processor system may give no indication of serious errors which would appear on multiprocessor systems, and visa versa. Thus test cases should include a varying number of processors.
- For programs which create a varying number of threads according to workload, tests which don't span the full range of possibilities may miss serious errors.

1.2.3. Getting a head start

Although it might appear that multithreaded programs are inherently unreliable, many reliable multithreaded programs do exist. Multithreading techniques are known which lead to reliable programs.

Design patterns for reliable multithreaded programs, including the important *monitor* pattern, are presented in *Pattern−Oriented Software Architecture Volume 2 − Patterns for Concurrent and Networked Objects*[SchmidtStalRohnertBuschmann]. Many important multithreading programming considerations (independent of threading library) are discussed in *Programming with POSIX Threads*[Butenhof97].

Doing some reading before attempting multithreaded designs will give you a head start toward reliable multithreaded programs.

1.3. C++ Standard Library usage in multithreaded programs

1.3.1. Runtime libraries

Warning: Multithreaded programs such as those using **Boost.Threads** must link to thread−safe versions of all runtime libraries used by the program, including the runtime library for the C++ Standard Library. Failure to do so will cause race conditions to occur when multiple threads simultaneously execute runtime library functions for `new`, `delete`, or other language features which imply shared state.

1.3.2. Potentially non−thread−safe functions

Certain C++ Standard Library functions inherited from C are particular problems because they hold internal state between calls:

- `rand`
- `strtok`
- `asctime`
- `ctime`
- `gmtime`
- `localtime`

It is possible to write thread−safe implementations of these by using thread specific storage (see boost::thread_specific_ptr), and several C++ compiler vendors do just that. The technique is well−know and is explained in [Butenhof97].

But at least one vendor (HP−UX) does not provide thread−safe implementations of the above functions in their otherwise thread−safe runtime library. Instead they provide replacement functions with different names and arguments.

Recommendation: For the most portable, yet thread−safe code, use Boost replacements for the problem functions. See the Boost Random Number Library and Boost Tokenizer Library.

1.4. Common guarantees for all Boost.Threads components

1.4.1. Exceptions

Boost.Threads destructors never throw exceptions. Unless otherwise specified, other **Boost.Threads** functions that do not have an exception−specification may throw implementation−defined exceptions.

In particular, **Boost.Threads** reports failure to allocate storage by throwing an exception of type `std::bad_alloc` or a class derived from `std::bad_alloc`, failure to obtain thread resources other than memory by throwing an exception of type boost::thread_resource_error, and certain lock related failures by throwing an exception of type boost::lock_error.

Rationale: Follows the C++ Standard Library practice of allowing all functions except destructors or other specified functions to throw exceptions on errors.

1.4.2. NonCopyable requirement

Boost.Threads classes documented as meeting the NonCopyable requirement disallow copy construction and copy assignment. For the sake of exposition, the synopsis of such classes show private derivation from boost::noncopyable. Users should not depend on this derivation, however, as implementations are free to meet the NonCopyable requirement in other ways.

2. Design

With client/server and three−tier architectures becoming common place in today's world, it's becoming increasingly important for programs to be able to handle parallel processing. Modern day operating systems usually provide some support for this through native thread APIs. Unfortunately, writing portable code that makes use of parallel processing in C++ is made very difficult by a lack of a standard interface for these native APIs. Further, these APIs are almost universally C APIs and fail to take advantage of C++'s strengths, or to address concepts unique to C++, such as exceptions.

The **Boost.Threads** library is an attempt to define a portable interface for writing parallel processes in C++.

2.1. Goals

The **Boost.Threads** library has several goals that should help to set it apart from other solutions. These goals are listed in order of precedence with full descriptions below.

Portability
: **Boost.Threads** was designed to be highly portable. The goal is for the interface to be easily implemented on any platform that supports threads, and possibly even on platforms without native thread support.

Safety
: **Boost.Threads** was designed to be as safe as possible. Writing thread−safe code is very difficult and successful libraries must strive to insulate the programmer from dangerous constructs as much as possible. This is accomplished in several ways:

  - C++ language features are used to make correct usage easy (if possible) and error−prone usage impossible or at least more difficult. For example, see the Mutex and Lock designs, and note how they interact.
  - Certain traditional concurrent programming features are considered so error−prone that they are not provided at all. For example, see Section 4.5, Rationale for not providing *Event Variables* .
  - Dangerous features, or features which may be misused, are identified as such in the documentation to make users aware of potential pitfalls.

Flexibility
: **Boost.Threads** was designed to be flexible. This goal is often at odds with *safety*. When functionality might be compromised by the desire to keep the interface safe, **Boost.Threads** has been designed to provide the functionality, but to make it's use prohibitive for general use. In other words, the interfaces have been designed such that it's usually obvious when something is unsafe, and the documentation is written to explain why.

Efficiency

Boost.Threads was designed to be as efficient as possible. When building a library on top of another library there is always a danger that the result will be so much slower than the "native" API that programmers are inclined to ignore the higher level API. **Boost.Threads** was designed to minimize the chances of this occurring. The interfaces have been crafted to allow an implementation the greatest chance of being as efficient as possible. This goal is often at odds with the goal for *safety*. Every effort was made to ensure efficient implementations, but when in conflict *safety* has always taken precedence.

2.2. Iterative Phases

Another goal of **Boost.Threads** was to take a dynamic, iterative approach in its development. The computing industry is still exploring the concepts of parallel programming. Most thread libraries supply only simple primitive concepts for thread synchronization. These concepts are very simple, but it is very difficult to use them safely or to provide formal proofs for constructs built on top of them. There has been a lot of research into other concepts, such as in "Communicating Sequential Processes." **Boost.Threads** was designed in iterative steps, with each step providing the building blocks necessary for the next step and giving the researcher the tools necessary to explore new concepts in a portable manner.

Given the goal of following a dynamic, iterative approach **Boost.Threads** shall go through several growth cycles. Each phase in its development shall be roughly documented here.

2.3. Phase 1, Synchronization Primitives

Boost is all about providing high quality libraries with implementations for many platforms. Unfortunately, there's a big problem faced by developers wishing to supply such high quality libraries, namely thread−safety. The C++ standard doesn't address threads at all, but real world programs often make use of native threading support. A portable library that doesn't address the issue of thread−safety is therefore not much help to a programmer who wants to use the library in his multithreaded application. So there's a very great need for portable primitives that will allow the library developer to create thread−safe implementations. This need far out weighs the need for portable methods to create and manage threads.

Because of this need, the first phase of **Boost.Threads** focuses solely on providing portable primitive concepts for thread synchronization. Types provided in this phase include the boost::mutex, boost::try_mutex, boost::timed_mutex, boost::recursive_mutex, boost::recursive_try_mutex, boost::recursive_timed_mutex, and boost::lock_error. These are considered the "core" synchronization primitives, though there are others that will be added in later phases.

2.4. Phase 2, Thread Management and Thread Specific Storage

This phase addresses the creation and management of threads and provides a mechanism for thread specific storage (data associated with a thread instance). Thread management is a tricky issue in C++, so this phase addresses only the basic needs of multithreaded program. Later phases are likely to add additional functionality in this area. This phase of **Boost.Threads** adds the boost::thread and boost::thread_specific_ptr types. With these additions the **Boost.Threads** library can be considered minimal but complete.

2.5. The Next Phase

The next phase will address more advanced synchronization concepts, such as read/write mutexes and barriers.

3. Concepts

Boost.Threads currently supports two types of mutex concepts: ordinary Mutexes, which allow only one thread at a time to access a resource, and Read/Write Mutexes, which allow only one thread at a time to access a resource when it is being modified (the "Write" part of Read/Write), but allows multiple threads to access a resource when it is only being referenced (the "Read" part of Read/Write).

3.1. Mutexes

Note

Certain changes to the mutexes and lock concepts are currently under discussion. In particular, the combination of the multiple lock concepts into a single lock concept is likely, and the combination of the multiple mutex concepts into a single mutex concept is also possible.
A mutex (short for mutual−exclusion) object is used to serialize access to a resource shared between multiple threads. The Mutex concept, with TryMutex and TimedMutex refinements, formalize the requirements. A model that implements Mutex and its refinements has two states: **locked** and **unlocked**. Before using a shared resource, a thread locks a **Boost.Threads** mutex object (an object whose type is a model of Mutex or one of it's refinements), ensuring thread−safe access to the shared resource. When use of the shared resource is complete, the thread unlocks the mutex object, allowing another thread to acquire the lock and use the shared resource.

Traditional C thread APIs, like POSIX threads or the Windows thread APIs, expose functions to lock and unlock a mutex object. This is dangerous since it's easy to forget to unlock a locked mutex. When the flow of control is complex, with multiple return points, the likelihood of forgetting to unlock a mutex object becomes even greater. When exceptions are thrown, it becomes nearly impossible to ensure that the mutex object is unlocked properly when using these traditional API's. The result is deadlock.

Many C++ threading libraries use a pattern known as *Scoped Locking*[SchmidtStalRohnertBuschmann] to free the programmer from the need to explicitly lock and unlock mutex objects. With this pattern, a Lock concept is employed where the lock object's constructor locks the associated mutex object and the destructor automatically does the unlocking. The **Boost.Threads** library takes this pattern to the extreme in that Lock concepts are the only way to lock and unlock a mutex object: lock and unlock functions are not exposed by any **Boost.Threads** mutex objects. This helps to ensure safe usage patterns, especially when code throws exceptions.

3.1.1. Locking Strategies

Every mutex object follows one of several locking strategies. These strategies define the semantics for the locking operation when the calling thread already owns a lock on the mutex object.

3.1.1.1. Recursive Locking Strategy

With a recursive locking strategy, when a thread attempts to acquire a lock on the mutex object for which it already owns a lock, the operation is successful. Note the distinction between a thread, which may have multiple locks outstanding on a recursive mutex object, and a lock object, which even for a recursive mutex object cannot have any of its lock functions called multiple times without first calling unlock.

Internally a lock count is maintained and the owning thread must unlock the mutex object the same number of times that it locked it before the mutex object's state returns to unlocked. Since mutex objects in **Boost.Threads** expose locking functionality only through lock concepts, a thread will always unlock a mutex object the same number of times that it locked it. This helps to eliminate a whole set of errors typically found in traditional C style thread APIs.

Classes boost::recursive_mutex, boost::recursive_try_mutex and boost::recursive_timed_mutex use this locking strategy.

3.1.1.2. Checked Locking Strategy

With a checked locking strategy, when a thread attempts to acquire a lock on the mutex object for which the thread already owns a lock, the operation will fail with some sort of error indication. Further, attempts by a thread to unlock a mutex object that was not locked by the thread will also return some sort of error indication. In **Boost.Threads**, an exception of type boost::lock_error would be thrown in these cases.

Boost.Threads does not currently provide any mutex objects that use this strategy.

3.1.1.3. Unchecked Locking Strategy

With an unchecked locking strategy, when a thread attempts to acquire a lock on a mutex object for which the thread already owns a lock the operation will deadlock. In general this locking strategy is less safe than a checked or recursive strategy, but it's also a faster strategy and so is employed by many libraries.

Boost.Threads does not currently provide any mutex objects that use this strategy.

3.1.1.4. Unspecified Locking Strategy

With an unspecified locking strategy, when a thread attempts to acquire a lock on a mutex object for which the thread already owns a lock the operation results in undefined behavior.

In general a mutex object with an unspecified locking strategy is unsafe, and it requires programmer discipline to use the mutex object properly. However, this strategy allows an implementation to be as fast as possible with no restrictions on its implementation. This is especially true for portable implementations that wrap the native threading support of a platform. For this reason, the classes boost::mutex, boost::try_mutex and boost::timed_mutex use this locking strategy despite the lack of safety.

3.1.2. Scheduling Policies

Every mutex object follows one of several scheduling policies. These policies define the semantics when the mutex object is unlocked and there is more than one thread waiting to acquire a lock. In other words, the policy defines which waiting thread shall acquire the lock.

3.1.2.1. FIFO Scheduling Policy

With a FIFO ("First In First Out") scheduling policy, threads waiting for the lock will acquire it in a first−come−first−served order. This can help prevent a high priority thread from starving lower priority threads that are also waiting on the mutex object's lock.

3.1.2.2. Priority Driven Policy

With a Priority Driven scheduling policy, the thread with the highest priority acquires the lock. Note that this means that low−priority threads may never acquire the lock if the mutex object has high contention and there is always at least one high−priority thread waiting. This is known as thread starvation. When multiple threads of the same priority are waiting on the mutex object's lock one of the other scheduling priorities will determine which thread shall acquire the lock.

3.1.2.3. Unspecified Policy

The mutex object does not specify a scheduling policy. In order to ensure portability, all **Boost.Threads** mutex objects use an unspecified scheduling policy.

3.1.3. Mutex Concepts

3.1.3.1. Mutex Concept

A Mutex object has two states: locked and unlocked. Mutex object state can only be determined by a lock object meeting the appropriate lock concept requirements and constructed for the Mutex object.

A Mutex is NonCopyable.

For a Mutex type `M` and an object `m` of that type, the following expressions must be well−formed and have the indicated effects.

Table 9.1. Mutex Expressions

Expression	Effects
M m;	Constructs a mutex object m. Postcondition: m is unlocked.
(&m)−>~M();	Precondition: m is unlocked. Destroys a mutex object m.
M::scoped_lock	A model of ScopedLock

3.1.3.2. TryMutex Concept

A TryMutex is a refinement of Mutex. For a TryMutex type `M` and an object `m` of that type, the following expressions must be well−formed and have the indicated effects.

Table 9.2. TryMutex Expressions

Expression	Effects
M::scoped_try_lock	A model of ScopedTryLock

3.1.3.3. TimedMutex Concept

A TimedMutex is a refinement of TryMutex. For a TimedMutex type `M` and an object `m` of that type, the following expressions must be well−formed and have the indicated effects.

Table 9.3. TimedMutex Expressions

Expression	Effects
M::scoped_timed_lock	A model of ScopedTimedLock

3.1.4. Mutex Models

Boost.Threads currently supplies six models of Mutex and its refinements.

Table 9.4. Mutex Models

Concept	Refines	Models
Mutex		boost::mutex boost::recursive_mutex
TryMutex	Mutex	boost::try_mutex boost::recursive_try_mutex
TimedMutex	TryMutex	boost::timed_mutex boost::recursive_timed_mutex

3.1.5. Lock Concepts

A lock object provides a safe means for locking and unlocking a mutex object (an object whose type is a model of Mutex or one of its refinements). In other words they are an implementation of the *Scoped Locking*[SchmidtStalRohnertBuschmann] pattern. The ScopedLock, ScopedTryLock, and ScopedTimedLock concepts formalize the requirements.

Lock objects are constructed with a reference to a mutex object and typically acquire ownership of the mutex object by setting its state to locked. They also ensure ownership is relinquished in the destructor. Lock objects also expose functions to query the lock status and to manually lock and unlock the mutex object.

Lock objects are meant to be short lived, expected to be used at block scope only. The lock objects are not thread−safe. Lock objects must maintain state to indicate whether or not they've been locked and this state is not protected by any synchronization concepts. For this reason a lock object should never be shared between multiple threads.

3.1.5.1. Lock Concept

For a Lock type `L` and an object `lk` and const object `clk` of that type, the following expressions must be well−formed and have the indicated effects.

Table 9.5. Lock Expressions

Expression	Effects
`(&lk)->~L();`	`if (locked()) unlock();`
`(&clk)->operator const void*()`	Returns type void*, non−zero if the associated mutex object has been locked by `clk`, otherwise 0.
`clk.locked()`	Returns a `bool`, `(&clk)->operator const void*() != 0`
`lk.lock()`	Throws boost::lock_error if `locked()`. If the associated mutex object is already locked by some other thread, places the current thread in the Blocked state until the associated mutex is unlocked, after which the current thread is placed in the Ready state, eventually to be returned to the Running state. If the associated mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated mutex object. Postcondition: `locked() == true`
`lk.unlock()`	Throws boost::lock_error if `!locked()`. Unlocks the associated mutex. Postcondition: `!locked()`

3.1.5.2. ScopedLock Concept

A ScopedLock is a refinement of Lock. For a ScopedLock type `L` and an object `lk` of that type, and an object `m` of a type meeting the Mutex requirements, and an object `b` of type `bool`, the following expressions must be well−formed and have the indicated effects.

Table 9.6. ScopedLock Expressions

Expression	Effects
`L lk(m);`	Constructs an object `lk`, and associates mutex object `m` with it, then calls `lock()`
`L lk(m,b);`	Constructs an object `lk`, and associates mutex object `m` with it, then if `b`, calls `lock()`

3.1.5.3. TryLock Concept

A TryLock is a refinement of Lock. For a TryLock type `L` and an object `lk` of that type, the following expressions must be well−formed and have the indicated effects.

Table 9.7. TryLock Expressions

Expression	Effects
`lk.try_lock()`	Throws boost::lock_error if locked(). Makes a non−blocking attempt to lock the associated mutex object, returning `true` if the lock attempt is successful, otherwise `false`. If the associated mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated mutex object.

3.1.5.4. ScopedTryLock Concept

A ScopedTryLock is a refinement of TryLock. For a ScopedTryLock type `L` and an object `lk` of that type, and an object `m` of a type meeting the TryMutex requirements, and an object `b` of type `bool`, the following expressions must be well−formed and have the indicated effects.

Table 9.8. ScopedTryLock Expressions

Expression	Effects
`L lk(m);`	Constructs an object `lk`, and associates mutex object `m` with it, then calls `try_lock()`
`L lk(m,b);`	Constructs an object `lk`, and associates mutex object `m` with it, then if `b`, calls `lock()`

3.1.5.5. TimedLock Concept

A TimedLock is a refinement of TryLock. For a TimedLock type `L` and an object `lk` of that type, and an object `t` of type boost::xtime, the following expressions must be well−formed and have the indicated effects.

Table 9.9. TimedLock Expressions

Expression	Effects
`lk.timed_lock(t)`	Throws boost::lock_error if locked(). Makes a blocking attempt to lock the associated mutex object, and returns `true` if successful within the specified time `t`, otherwise `false`. If the associated mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated mutex object.

3.1.5.6. ScopedTimedLock Concept

A ScopedTimedLock is a refinement of TimedLock. For a ScopedTimedLock type `L` and an object `lk` of that type, and an object `m` of a type meeting the TimedMutex requirements, and an object `b` of type `bool`, and an object `t` of type boost::xtime, the following expressions must be well−formed and have the indicated effects.

Table 9.10. ScopedTimedLock Expressions

Expression	Effects
`L lk(m,t);`	Constructs an object `lk`, and associates mutex object `m` with it, then calls `timed_lock(t)`
`L lk(m,b);`	Constructs an object `lk`, and associates mutex object `m` with it, then if `b`, calls `lock()`

3.1.6. Lock Models

Boost.Threads currently supplies twelve models of Lock and its refinements.

Table 9.11. Lock Models

Concept	Refines	Models
Lock		
ScopedLock	Lock	boost::mutex::scoped_lock boost::recursive_mutex::scoped_lock boost::try_mutex::scoped_lock boost::recursive_try_mutex::scoped_lock boost::timed_mutex::scoped_lock boost::recursive_timed_mutex::scoped_lock
TryLock	Lock	
ScopedTryLock	TryLock	boost::try_mutex::scoped_try_lock boost::recursive_try_mutex::scoped_try_lock boost::timed_mutex::scoped_try_lock boost::recursive_timed_mutex::scoped_try_lock
TimedLock	TryLock	
ScopedTimedLock	TimedLock	boost::timed_mutex::scoped_timed_lock boost::recursive_timed_mutex::scoped_timed_lock

3.2. Read/Write Mutexes

Note

Since the read/write mutex and related classes are new, both interface and implementation are liable to change in future releases of **Boost.Threads**. The lock concepts and lock promotion and demotion in particular are still under discussion and very likely to change.
A read/write mutex (short for reader/writer mutual−exclusion) object is used to serialize access to a resource shared between multiple threads, where multiple "readers" can share simultaneous access, but "writers" require exclusive access. The ReadWriteMutex concept, with TryReadWriteMutex and TimedReadWriteMutex refinements formalize the requirements. A model that implements ReadWriteMutex and its refinements has three states: **read−locked**, **write−locked**, and **unlocked**. Before reading from a shared resource, a thread **read−locks** a **Boost.Threads** read/write mutex object (an object whose type is a model of ReadWriteMutex or one of it's refinements), ensuring thread−safe access for reading from the shared resource. Before writing to a shared resource, a thread **write−locks** a **Boost.Threads** read/write mutex object (an object whose type is a model of ReadWriteMutex or one of it's refinements), ensuring thread−safe access for altering the shared resource. When use of the shared resource is complete, the thread unlocks the mutex object, allowing another thread to acquire the lock and use the shared resource.

Traditional C thread APIs that provide read/write mutex primitives (like POSIX threads) expose functions to lock and unlock a mutex object. This is dangerous since it's easy to forget to unlock a locked mutex. When the flow of control is complex, with multiple return points, the likelihood of forgetting to unlock a mutex object becomes even greater. When exceptions are thrown, it becomes nearly impossible to ensure that the mutex object is unlocked properly when using these traditional API's. The result is deadlock.

Many C++ threading libraries use a pattern known as *Scoped Locking*[SchmidtStalRohnertBuschmann] to free the programmer from the need to explicitly lock and unlock read/write mutex objects. With this pattern, a Read/Write Lock concept is employed where the lock object's constructor locks the associated read/write mutex object and the destructor automatically does the unlocking. The **Boost.Threads** library takes this pattern to the extreme in that Read/Write Lock concepts are the only way to lock and unlock a read/write mutex object: lock and unlock functions are not exposed by any **Boost.Threads** read/write mutex objects. This helps to ensure safe usage patterns, especially when code throws exceptions.

3.2.1. Locking Strategies

Every read/write mutex object follows one of several locking strategies. These strategies define the semantics for the locking operation when the calling thread already owns a lock on the read/write mutex object.

3.2.1.1. Recursive Locking Strategy

With a recursive locking strategy, when a thread attempts to acquire a lock on a read/write mutex object for which it already owns a lock, the operation is successful, except in the case where a thread holding a read−lock attempts to obtain a write lock, in which case a boost::lock_error exception will be thrown. Note the distinction between a thread, which may have multiple locks outstanding on a recursive read/write mutex object, and a lock object, which even for a recursive read/write mutex object cannot have any of its lock functions called multiple times without first calling unlock.

Lock Type Held	Lock Type Requested	Action
read−lock	read−lock	Grant the read−lock immediately
read−lock	write−lock	If this thread is the only holder of the read−lock, grants the write lock immediately. Otherwise throws a boost::lock_error exception.
write−locked	read−lock	Grants the (additional) read−lock immediately.
write−locked	write−lock	Grant the write−lock immediately

Internally a lock count is maintained and the owning thread must unlock the mutex object the same number of times that it locked it before the mutex object's state returns to unlocked. Since mutex objects in **Boost.Threads** expose locking functionality only through lock concepts, a thread will always unlock a mutex object the same number of times that it locked it. This helps to eliminate a whole set of errors typically found in traditional C style thread APIs.

Boost.Threads does not currently provide any read/write mutex objects that use this strategy. A successful implementation of this locking strategy may require thread identification.

3.2.1.2. Checked Locking Strategy

With a checked locking strategy, when a thread attempts to acquire a lock on the mutex object for which the thread already owns a lock, the operation will fail with some sort of error indication, except in the case of multiple read−lock acquisition which is a normal operation for ANY ReadWriteMutex. Further, attempts by a thread to unlock a mutex that was not locked by the thread will also return some sort of error indication. In **Boost.Threads**, an exception of type boost::lock_error would be thrown in these cases.

Lock Type Held	Lock Type Requested	Action
read−lock	read−lock	Grant the read−lock immediately
read−lock	write−lock	Throw boost::lock_error
write−locked	read−lock	Throw boost::lock_error
write−locked	write−lock	Throw boost::lock_error

Boost.Threads does not currently provide any read/write mutex objects that use this strategy. A successful implementation of this locking strategy may require thread identification.

3.2.1.3. Unchecked Locking Strategy

With an unchecked locking strategy, when a thread attempts to acquire a lock on the read/write mutex object for which the thread already owns a lock, the operation will deadlock. In general this locking strategy is less safe than a checked or recursive strategy, but it can be a faster strategy and so is employed by many libraries.

Lock Type Held	Lock Type Requested	Action
read−lock	read−lock	Grant the read−lock immediately
read−lock	write−lock	Deadlock
write−locked	read−lock	Deadlock
write−locked	write−lock	Deadlock

Boost.Threads does not currently provide any mutex objects that use this strategy. For ReadWriteMutexes on platforms that contain natively recursive synchronization primitives, implementing a guaranteed−deadlock can actually involve extra work, and would likely require thread identification.

3.2.1.4. Unspecified Locking Strategy

With an unspecified locking strategy, when a thread attempts to acquire a lock on a read/write mutex object for which the thread already owns a lock, the operation results in undefined behavior. When a read/write mutex object has an unspecified locking strategy the programmer must assume that the read/write mutex object instead uses an unchecked strategy as the worse case, although some platforms may exhibit a mix of unchecked and recursive behavior.

Lock Type Held	Lock Type Requested	Action

read−lock	read−lock	Grant the read−lock immediately
read−lock	write−lock	Undefined, but generally deadlock
write−locked	read−lock	Undefined, but generally deadlock
write−locked	write−lock	Undefined, but generally deadlock

In general a read/write mutex object with an unspecified locking strategy is unsafe, and it requires programmer discipline to use the read/write mutex object properly. However, this strategy allows an implementation to be as fast as possible with no restrictions on its implementation. This is especially true for portable implementations that wrap the native threading support of a platform. For this reason, the classes read_write_mutex, try_read_write_mutex, and timed_read_write_mutex use this locking strategy despite the lack of safety.

3.2.1.5. Thread Identification

ReadWriteMutexes can support specific Locking Strategies (recursive and checked) which help to detect and protect against self−deadlock. Self−deadlock can occur when a holder of a locked ReadWriteMutex attempts to obtain another lock. Given an implemention *I* which is susceptible to self−deadlock but otherwise correct and efficient, a recursive or checked implementation *Ir* or *Ic* can use the same basic implementation, but make special checks against self−deadlock by tracking the identities of thread(s) currently holding locks. This approach makes deadlock detection othrogonal to the basic ReadWriteMutex implementaion.

Alternatively, a different basic implementation for ReadWriteMutex concepts, *I'* (I−Prime) may exist which uses recursive or checked versions of synchronization primitives to produce a recursive or checked ReadWriteMutex while still providing flexibility in terms of Scheduling Policies.

Please refer to the **Boost.Threads**read/write mutex concept documentation for a discussion of locking strategies. The read/write mutex supports only the unspecified locking strategy. ReadWriteMutexes are parameterized on a Mutex type which they use to control write−locking and access to internal state.

3.2.1.6. Lock Promotion

ReadWriteMutexes can support lock promotion, where a mutex which is in the read−locked state transitions to a write−locked state without releasing the lock. Lock promotion can be tricky to implement; for instance, extra care must be taken to ensure that only one thread holding a read−lock can block awaiting promotion at any given time. If more than one read−lock holder is allowed to enter a blocked state while waiting to be promoted, deadlock will result since both threads will be waiting for the other to release their read−lock.

Currently, **Boost.Threads** supports lock promotion through `promote()`, `try_promote()`, and `timed_promote()` operations.

3.2.1.7. Lock Demotion

ReadWriteMutexes can support lock demotion, where a mutex which is in the write−locked state transitions to a read−locked state without releasing the lock. Since by definition only one thread at a time may hold a write−lock, the problem with deadlock that can occur during lock promotion is not a problem for lock demotion.

Currently, **Boost.Threads** supports lock demotion through `demote()`, `try_demote()`, and `timed_demote()` operations.

3.2.2. Scheduling Policies

Every read/write mutex object follows one of several scheduling policies. These policies define the semantics when the mutex object is unlocked and there is more than one thread waiting to acquire a lock. In other words, the policy defines which waiting thread shall acquire the lock. For a read/write mutex, it is particularly important to define the behavior when threads are requesting both read and write access simultaneously. This will be referred to as "inter−class scheduling" because it describes the scheduling between two classes of threads (those waiting for a read lock and those waiting for a write lock).

For some types of inter−class scheduling, an "intra−class" scheduling policy can also be defined that will describe the order in which waiting threads of the same class (i.e., those waiting for the same type of lock) will acquire the thread.

3.2.2.1. Inter−Class Scheduling Policies

3.2.2.1.1. ReaderPriority

With ReaderPriority scheduling, any pending request for a read−lock will have priority over a pending request for a write−lock, irrespective of the current lock state of the read/write mutex, and irrespective of the relative order that the pending requests arrive.

Current mutex state	Request Type	Action
unlocked	read−lock	Grant the read−lock immediately
read−locked	read−lock	Grant the additional read−lock immediately.
write−locked	read−lock	Wait to acquire the lock until the thread holding the write−lock releases its lock (or until the specified time, if any). A read−lock will be granted to all pending readers before any other thread can acquire a write−lock. TODO: try−lock, timed−lock.
unlocked	write−lock	Grant the write−lock immediately, if and only if there are no pending read−lock requests. TODO: try−lock, timed−lock.
read−locked	write−lock	Wait to acquire the lock until all threads holding read−locks release their locks **AND** no requests for read−locks exist. If other write−lock requests exist, the lock is granted in accordance with the intra−class scheduling policy. TODO: try−lock, timed−lock.
write−locked	write−lock	Wait to acquire the lock until the thread holding the write−lock releases its lock **AND** no requests for read−locks exist. If other write−lock requests exist, the lock is granted in accordance with the intra−class scheduling policy. TODO: try−lock, timed−lock.
read−locked	promote	TODO
write−locked	demote	TODO

3.2.2.1.2. WriterPriority

With WriterPriority scheduling, any pending request for a write−lock will have priority over a pending request for a read−lock, irrespective of the current lock state of the read/write mutex, and irrespective of the relative order that the pending requests arrive.

Current mutex state	Request Type	Action
unlocked	read−lock	Grant the read−lock immediately.
read−locked	read−lock	Grant the additional read−lock immediately, **IF** no outstanding requests for a write−lock

		exist; otherwise TODO. TODO: try−lock, timed−lock.
write−locked	read−lock	Wait to acquire the lock until the thread holding the write−lock releases its lock. The read lock will be granted once no other outstanding write−lock requests exist. TODO: try−lock, timed−lock.
unlocked	write−lock	Grant the write−lock immediately.
read−locked	write−lock	Wait to acquire the lock until all threads holding read−locks release their locks. If other write−lock requests exist, the lock is granted in accordance with the intra−class scheduling policy. This request will be granted before any new read−lock requests are granted. TODO: try−lock, timed−lock.
write−locked	write−lock	Wait to acquire the lock until the thread holding the write−lock releases its lock. If other write−lock requests exist, the lock is granted in accordance with the intra−class scheduling policy. This request will be granted before any new read−lock requests are granted. TODO: try−lock, timed−lock.
read−locked	promote	TODO
write−locked	demote	TODO

3.2.2.1.3. AlternatingPriority/ManyReads

With AlternatingPriority/ManyReads scheduling, reader or writer starvation is avoided by alternatively granting read or write access when pending requests exist for both types of locks. Outstanding read−lock requests are treated as a group when it is the "readers' turn"

Current mutex state	Request Type	Action
unlocked	read−lock	Grant the read−lock immediately.
read−locked	read−lock	Grant the additional read−lock immediately, **IF** no outstanding requests for a write−lock exist. If outstanding write−lock requests exist, this lock will not be granted until at least one of the write−locks is granted and released. If other read−lock requests exist, all read−locks will be granted as a group. TODO: try−lock, timed−lock.
write−locked	read−lock	Wait to acquire the lock until the thread holding the write−lock releases its lock. If other outstanding write−lock requests exist, they will have to wait until all current read−lock requests are serviced. TODO: try−lock, timed−lock.
unlocked	write−lock	Grant the write−lock immediately.
read−locked	write−lock	Wait to acquire the lock until all threads holding read−locks release their locks. If other write−lock requests exist, this lock will be granted to one of them in accordance with the intra−class scheduling policy.

		TODO: try−lock, timed−lock.
write−locked	write−lock	Wait to acquire the lock until the thread holding the write−lock releases its lock. If other outstanding read−lock requests exist, this lock will not be granted until all of the currently waiting read−locks are granted and released. If other write−lock requests exist, this lock will be granted in accordance with the intra−class scheduling policy. TODO: try−lock, timed−lock.
read−locked	promote	TODO
write−locked	demote	TODO

3.2.2.1.4. AlternatingPriority/SingleRead

With AlternatingPriority/SingleRead scheduling, reader or writer starvation is avoided by alternatively granting read or write access when pending requests exist for both types of locks. Outstanding read−lock requests are services one at a time when it is the "readers' turn"

Current mutex state	Request Type	Action
unlocked	read−lock	Grant the read−lock immediately.
read−locked	read−lock	Grant the additional read−lock immediately, **IF** no outstanding requests for a write−lock exist. If outstanding write−lock requests exist, this lock will not be granted until at least one of the write−locks is granted and released. TODO: try−lock, timed−lock.
write−locked	read−lock	Wait to acquire the lock until the thread holding the write−lock releases its lock. If other outstanding write−lock requests exist, exactly one read−lock request will be granted before the next write−lock is granted. TODO: try−lock, timed−lock.
unlocked	write−lock	Grant the write−lock immediately.
read−locked	write−lock	Wait to acquire the lock until all threads holding read−locks release their locks. If other write−lock requests exist, this lock will be granted to one of them in accordance with the intra−class scheduling policy.
write−locked	write−lock	Wait to acquire the lock until the thread holding the write−lock releases its lock. If other outstanding read−lock requests exist, this lock can not be granted until exactly one read−lock request is granted and released. If other write−lock requests exist, this lock will be granted in accordance with the intra−class scheduling policy. TODO: try−lock, timed−lock.
read−locked	promote	TODO
write−locked	demote	TODO

3.2.2.2. Intra–Class Scheduling Policies

Please refer to Section 3.1.2, Scheduling Policies for a discussion of mutex scheduling policies, which are identical to read/write mutex intra–class scheduling policies.

For threads waiting to obtain write–locks, the read/write mutex supports only the Unspecified intra–class scheduling policy. That is, given a set of threads waiting for write–locks, the order, relative to one another, in which they receive the write–lock is unspecified.

For threads waiting to obtain read–locks, the read/write mutex supports only the Unspecified intra–class scheduling policy. That is, given a set of threads waiting for read–locks, the order, relative to one another, in which they receive the read–lock is unspecified.

3.2.3. Mutex Concepts

3.2.3.1. ReadWriteMutex Concept

A ReadWriteMutex object has three states: read–locked, write–locked, and unlocked. ReadWriteMutex object state can only be determined by a lock object meeting the appropriate lock concept requirements and constructed for the ReadWriteMutex object.

A ReadWriteMutex is NonCopyable.

For a ReadWriteMutex type `M`, and an object `m` of that type, the following expressions must be well–formed and have the indicated effects.

Table 9.12. ReadWriteMutex Expressions

Expression	Effects
`M m;`	Constructs a read/write mutex object `m`. Post–condition: `m` is unlocked.
`(&m)->~M();`	Precondition: `m` is unlocked. Destroys a read/write mutex object `m`.
`M::scoped_read_write_lock`	A type meeting the ScopedReadWriteLock requirements.
`M::scoped_read_lock`	A type meeting the ScopedLock requirements.
`M::scoped_write_lock`	A type meeting the ScopedLock requirements.

3.2.3.2. TryReadWriteMutex Concept

A TryReadWriteMutex is a refinement of ReadWriteMutex. For a TryReadWriteMutex type `M` and an object `m` of that type, the following expressions must be well–formed and have the indicated effects.

Table 9.13. TryReadWriteMutex Expressions

Expression	Effects
`M::scoped_try_read_write_lock`	A type meeting the ScopedTryReadWriteLock requirements.
`M::scoped_try_read_lock`	A type meeting the ScopedTryLock requirements.
`M::scoped_try_write_lock`	A type meeting the ScopedTryLock requirements.

3.2.3.3. TimedReadWriteMutex Concept

A TimedReadWriteMutex is a refinement of TryReadWriteMutex. For a TimedReadWriteMutex type `M` and an object `m` of that type the following expressions must be well–formed and have the indicated effects.

Table 9.14. TimedReadWriteMutex Expressions

Expression	Effects
`M::scoped_timed_read_write_lock`	A type meeting the ScopedTimedReadWriteLock requirements.
`M::scoped_timed_read_lock`	A type meeting the ScopedTimedLock requirements.
`M::scoped_timed_write_lock`	A type meeting the ScopedTimedLock requirements.

3.2.4. Mutex Models

Boost.Threads currently supplies three models of ReadWriteMutex and its refinements.

Table 9.15. Mutex Models

Concept	Refines	Models
ReadWriteMutex		boost::read_write_mutex
TryReadWriteMutex	ReadWriteMutex	boost::try_read_write_mutex
TimedReadWriteMutex	TryReadWriteMutex	boost::timed_read_write_mutex

3.2.5. Lock Concepts

A read/write lock object provides a safe means for locking and unlocking a read/write mutex object (an object whose type is a model of ReadWriteMutex or one of its refinements). In other words they are an implementation of the *Scoped Locking*[SchmidtStalRohnertBuschmann] pattern. The ScopedReadWriteLock, ScopedTryReadWriteLock, and ScopedTimedReadWriteLock concepts formalize the requirements.

Read/write lock objects are constructed with a reference to a read/write mutex object and typically acquire ownership of the read/write mutex object by setting its state to locked. They also ensure ownership is relinquished in the destructor. Lock objects also expose functions to query the lock status and to manually lock and unlock the read/write mutex object.

Read/write lock objects are meant to be short lived, expected to be used at block scope only. The read/write lock objects are not thread–safe. Read/write lock objects must maintain state to indicate whether or not they've been locked and this state is not protected by any synchronization concepts. For this reason a read/write lock object should never be shared between multiple threads.

3.2.5.1. ReadWriteLock Concept

For a read/write lock type `L` and an object `lk` and const object `clk` of that type, the following expressions must be well–formed and have the indicated effects.

Table 9.16. ReadWriteLock Expressions

Expression	Effects

`(&lk)->~L();`	`if (locked()) unlock();`
`(&clk)->operator const void*()`	Returns type void*, non–zero if the associated read/write mutex object has been either read–locked or write–locked by `clk`, otherwise 0.
`clk.locked()`	Returns a `bool`, `(&clk)->operator const void*() != 0`
`clk.state()`	Returns an enumeration constant of type `read_write_lock_state`: `read_write_lock_state::read_locked` if the associated read/write mutex object has been read–locked by `clk`, `read_write_lock_state::write_locked` if it has been write–locked by `clk`, and `read_write_lock_state::unlocked` if has not been locked by `clk`.
`clk.read_locked()`	Returns a `bool`, `(&clk)->state() == read_write_lock_state::read_locked`.
`clk.write_locked()`	Returns a `bool`, `(&clk)->state() == read_write_lock_state::write_locked`.
`lk.read_lock()`	Throws boost::lock_error if `locked()`. If the associated read/write mutex object is already read–locked by some other thread, the effect depends on the inter–class scheduling policy of the associated read/write mutex: either immediately obtains an additional read–lock on the associated read/write mutex, or places the current thread in the Blocked state until the associated read/write mutex is unlocked, after which the current thread is placed in the Ready state, eventually to be returned to the Running state. If the associated read/write mutex object is already write–locked by some other thread, places the current thread in the Blocked state until the associated read/write mutex is unlocked, after which the current thread is placed in the Ready state, eventually to be returned to the Running state. If the associated read/write mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated read/write mutex object. Postcondition: `state() == read_write_lock_state::read_locked`
`lk.write_lock()`	Throws boost::lock_error if `locked()`. If the associated read/write mutex object is already locked by some other thread, places the current thread in the Blocked state until the associated read/write mutex is unlocked, after which the current thread is placed in the Ready state, eventually to be returned to the Running state. If the associated read/write mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated read/write mutex object. Postcondition: `state() == read_write_lock_state::write_locked`
`lk.demote()`	Throws boost::lock_error if `state() != read_write_lock_state::write_locked`. Converts the lock held on the associated read/write mutex object from a write–lock to a read–lock without releasing the lock. Postcondition: `state() == read_write_lock_state::read_locked`

`lk.promote()`	Throws boost::lock_error if `state() != read_write_lock_state::read_locked` or if the lock cannot be promoted because another lock on the same mutex is already waiting to be promoted. Makes a blocking attempt to convert the lock held on the associated read/write mutex object from a read–lock to a write–lock without releasing the lock.
`lk.unlock()`	Throws boost::lock_error if `!locked()`. Unlocks the associated read/write mutex. Postcondition: `!locked()`

3.2.5.2. ScopedReadWriteLock Concept

A ScopedReadWriteLock is a refinement of ReadWriteLock. For a ScopedReadWriteLock type `L` and an object `lk` of that type, and an object `m` of a type meeting the ReadWriteMutex requirements, and an object `s` of type `read_write_lock_state`, the following expressions must be well–formed and have the indicated effects.

Table 9.17. ScopedReadWriteLock Expressions

Expression	Effects
`L lk(m,s);`	Constructs an object `lk` and associates read/write mutex object `m` with it, then: if `s == read_write_lock_state::read_locked`, calls `read_lock()`; if `s==read_write_lock_state::write_locked`, calls `write_lock()`.

3.2.5.3. TryReadWriteLock Expressions

A TryReadWriteLock is a refinement of ReadWriteLock. For a TryReadWriteLock type `L` and an object `lk` of that type, the following expressions must be well–formed and have the indicated effects.

Table 9.18. TryReadWriteLock Expressions

Expression	Effects
`lk.try_read_lock()`	Throws boost::lock_error if locked(). Makes a non–blocking attempt to read–lock the associated read/write mutex object, returning `true` if the attempt is successful, otherwise `false`. If the associated read/write mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated read/write mutex object.
`lk.try_write_lock()`	Throws boost::lock_error if locked(). Makes a non–blocking attempt to write–lock the associated read/write mutex object, returning `true` if the attempt is successful, otherwise `false`. If the associated read/write mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated read/write mutex object.
`lk.try_demote()`	Throws boost::lock_error if `state() != read_write_lock_state::write_locked`.

	Makes a non−blocking attempt to convert the lock held on the associated read/write mutex object from a write−lock to a read−lock without releasing the lock, returning `true` if the attempt is successful, otherwise `false`.
`lk.try_promote()`	Throws boost::lock_error if `state() != read_write_lock_state::read_locked`. Makes a non−blocking attempt to convert the lock held on the associated read/write mutex object from a read−lock to a write−lock without releasing the lock, returning `true` if the attempt is successful, otherwise `false`.

3.2.5.4. ScopedTryReadWriteLock Expressions

A ScopedTryReadWriteLock is a refinement of TryReadWriteLock. For a ScopedTryReadWriteLock type `L` and an object `lk` of that type, and an object `m` of a type meeting the TryReadWriteMutex requirements, and an object `s` of type `read_write_lock_state`, and an object `b` of type `blocking_mode`, the following expressions must be well−formed and have the indicated effects.

Table 9.19. ScopedTryReadWriteLock Expressions

Expression	Effects
`L lk(m,s,b);`	Constructs an object `lk` and associates read/write mutex object `m` with it, then: if `s == read_write_lock_state::read_locked`, calls `read_lock()` if `b`, otherwise `try_read_lock()`; if `s==read_write_lock_state::write_locked`, calls `write_lock()` if `b`, otherwise `try_write_lock`.

3.2.5.5. TimedReadWriteLock Concept

A TimedReadWriteLock is a refinement of TryReadWriteLock. For a TimedReadWriteLock type `L` and an object `lk` of that type, and an object `t` of type boost::xtime, the following expressions must be well−formed and have the indicated effects.

Table 9.20. TimedReadWriteLock Expressions

Expression	Effects
`lk.timed_read_lock(t)`	Throws boost::lock_error if locked(). Makes a blocking attempt to read−lock the associated read/write mutex object, and returns `true` if successful within the specified time `t`, otherwise `false`. If the associated read/write mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated read/write mutex object.
`lk.timed_write_lock(t)`	Throws boost::lock_error if locked(). Makes a blocking attempt to write−lock the associated read/write mutex object, and returns `true` if successful within the specified time `t`, otherwise `false`. If the associated read/write mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated read/write mutex object.
`lk.timed_demote(t)`	Throws boost::lock_error if `state() != read_write_lock_state::write_locked`.

	Makes a blocking attempt to convert the lock held on the associated read/write mutex object from a write–lock to a read–lock without releasing the lock, returning `true` if the attempt is successful in the specified time `t`, otherwise `false`.
`lk.timed_promote(t)`	Throws boost::lock_error if `state() != read_write_lock_state::read_locked`. Makes a blocking attempt to convert the lock held on the associated read/write mutex object from a read–lock to a write–lock without releasing the lock, returning `true` if the attempt is successful in the specified time `t`, otherwise `false`.

3.2.5.6. ScopedTimedReadWriteLock Concept

A ScopedTimedReadWriteLock is a refinement of TimedReadWriteLock. For a ScopedTimedReadWriteLock type `L` and an object `lk` of that type, and an object `m` of a type meeting the TimedReadWriteMutex requirements, and an object `s` of type `read_write_lock_state`, and an object `t` of type boost::xtime, and an object `b` of type `blocking_mode`, the following expressions must be well–formed and have the indicated effects.

Table 9.21. ScopedTimedReadWriteLock Expressions

Expression	Effects
`L lk(m,s,b);`	Constructs an object `lk` and associates read/write mutex object `m` with it, then: if `s == read_write_lock_state::read_locked`, calls `read_lock()` if `b`, otherwise `try_read_lock()`; if `s==read_write_lock_state::write_locked`, calls `write_lock()` if `b`, otherwise `try_write_lock`.
`L lk(m,s,t);`	Constructs an object `lk` and associates read/write mutex object `m` with it, then: if `s == read_write_lock_state::read_locked`, calls `timed_read_lock(t)`; if `s==read_write_lock_state::write_locked`, calls `timed_write_lock(t)`.

3.2.6. Lock Models

Boost.Threads currently supplies six models of ReadWriteLock and its refinements.

Table 9.22. Lock Models

Concept	Refines	Models
ReadWriteLock		
ScopedReadWriteLock	ReadWriteLock	boost::read_write_mutex::scoped_read_write_lock boost::try_read_write_mutex::scoped_read_write_lock boost::timed_read_write_mutex::scoped_read_write_lock
TryReadWriteLock	ReadWriteLock	
ScopedTryReadWriteLock	TryReadWriteLock	boost::try_read_write_mutex::scoped_try_read_write_lock boost::timed_read_write_mutex::scoped_try_read_write_lock
TimedReadWriteLock	TryReadWriteLock	

ScopedTimedReadWriteLock	TimedReadWriteLock	boost::timed_read_write_mutex::scoped_timed_read_write_lock

4. Rationale

This page explains the rationale behind various design decisions in the **Boost.Threads** library. Having the rationale documented here should explain how we arrived at the current design as well as prevent future rehashing of discussions and thought processes that have already occurred. It can also give users a lot of insight into the design process required for this library.

4.1. Rationale for the Creation of Boost.Threads

Processes often have a degree of "potential parallelism" and it can often be more intuitive to design systems with this in mind. Further, these parallel processes can result in more responsive programs. The benefits for multithreaded programming are quite well known to most modern programmers, yet the C++ language doesn't directly support this concept.

Many platforms support multithreaded programming despite the fact that the language doesn't support it. They do this through external libraries, which are, unfortunately, platform specific. POSIX has tried to address this problem through the standardization of a "pthread" library. However, this is a standard only on POSIX platforms, so its portability is limited.

Another problem with POSIX and other platform specific thread libraries is that they are almost universally C based libraries. This leaves several C++ specific issues unresolved, such as what happens when an exception is thrown in a thread. Further, there are some C++ concepts, such as destructors, that can make usage much easier than what's available in a C library.

What's truly needed is C++ language support for threads. However, the C++ standards committee needs existing practice or a good proposal as a starting point for adding this to the standard.

The **Boost.Threads** library was developed to provide a C++ developer with a portable interface for writing multithreaded programs on numerous platforms. There's a hope that the library can be the basis for a more detailed proposal for the C++ standards committee to consider for inclusion in the next C++ standard.

4.2. Rationale for the Low Level Primitives Supported in Boost.Threads

The **Boost.Threads** library supplies a set of low level primitives for writing multithreaded programs, such as mutexes and condition variables. In fact, the first release of **Boost.Threads** supports only these low level primitives. However, computer science research has shown that use of these primitives is difficult since it's difficult to mathematically prove that a usage pattern is correct, meaning it doesn't result in race conditions or deadlocks. There are several algebras (such as CSP, CCS and Join calculus) that have been developed to help write provably correct parallel processes. In order to prove the correctness these processes must be coded using higher level abstractions. So why does **Boost.Threads** support the lower level concepts?

The reason is simple: the higher level concepts need to be implemented using at least some of the lower level concepts. So having portable lower level concepts makes it easier to develop the higher level concepts and will allow researchers to experiment with various techniques.

Beyond this theoretical application of higher level concepts, however, the fact remains that many multithreaded programs are written using only the lower level concepts, so they are useful in and of themselves, even if it's hard to prove that their usage is correct. Since many users will be familiar with these lower level concepts but unfamiliar with any of the higher level concepts, supporting the lower level concepts provides greater accessibility.

4.3. Rationale for the Lock Design

Programmers who are used to multithreaded programming issues will quickly note that the **Boost.Threads** design for mutex lock concepts is not thread−safe (this is clearly documented as well). At first this may seem like a serious design flaw. Why have a multithreading primitive that's not thread−safe itself?

A lock object is not a synchronization primitive. A lock object's sole responsibility is to ensure that a mutex is both locked and unlocked in a manner that won't result in the common error of locking a mutex and then forgetting to unlock it. This means that instances of a lock object are only going to be created, at least in theory, within block scope and won't be shared between threads. Only the mutex objects will be created outside of block scope and/or shared between threads. Though it's possible to create a lock object outside of block scope and to share it between threads, to do so would not be a typical usage (in fact, to do so would likely be an error). Nor are there any cases when such usage would be required.

Lock objects must maintain some state information. In order to allow a program to determine if a try_lock or timed_lock was successful the lock object must retain state indicating the success or failure of the call made in its constructor. If a lock object were to have such state and remain thread−safe it would need to synchronize access to the state information which would result in roughly doubling the time of most operations. Worse, since checking the state can occur only by a call after construction, we'd have a race condition if the lock object were shared between threads.

So, to avoid the overhead of synchronizing access to the state information and to avoid the race condition, the **Boost.Threads** library simply does nothing to make lock objects thread−safe. Instead, sharing a lock object between threads results in undefined behavior. Since the only proper usage of lock objects is within block scope this isn't a problem, and so long as the lock object is properly used there's no danger of any multithreading issues.

4.4. Rationale for NonCopyable Thread Type

Programmers who are used to C libraries for multithreaded programming are likely to wonder why **Boost.Threads** uses a noncopyable design for boost::thread. After all, the C thread types are copyable, and you often have a need for copying them within user code. However, careful comparison of C designs to C++ designs shows a flaw in this logic.

All C types are copyable. It is, in fact, not possible to make a noncopyable type in C. For this reason types that represent system resources in C are often designed to behave very similarly to a pointer to dynamic memory. There's an API for acquiring the resource and an API for releasing the resource. For memory we have pointers as the type and alloc/free for the acquisition and release APIs. For files we have FILE* as the type and fopen/fclose for the acquisition and release APIs. You can freely copy instances of the types but must manually manage the lifetime of the actual resource through the acquisition and release APIs.

C++ designs recognize that the acquisition and release APIs are error prone and try to eliminate possible errors by acquiring the resource in the constructor and releasing it in the destructor. The best example of such a design is the std::iostream set of classes which can represent the same resource as the FILE* type in C. A file is opened in the std::fstream's constructor and closed in its destructor. However, if an iostream were copyable it could lead to a file being closed twice, an obvious error, so the std::iostream types are noncopyable by design. This is the same design used by boost::thread, which is a simple and easy to understand design that's consistent with other C++ standard types.

During the design of boost::thread it was pointed out that it would be possible to allow it to be a copyable type if some form of "reference management" were used, such as ref−counting or ref−lists, and many argued for a boost::thread_ref design instead. The reasoning was that copying "thread" objects was a typical need in the C libraries, and so presumably would be in the C++ libraries as well. It was also thought that implementations could provide more efficient reference management than wrappers (such as boost::shared_ptr) around a noncopyable thread concept. Analysis of whether or not these arguments would hold true doesn't appear to bear them out. To illustrate the analysis we'll first provide pseudo−code illustrating the six typical usage patterns of a thread object.

4.4.1. 1. Use case: Simple creation of a thread.

```
void foo()
{
   create_thread(&bar);
}
```

4.4.2. 2. Use case: Creation of a thread that's later joined.

```
void foo()
{
   thread = create_thread(&bar);
   join(thread);
}
```

4.4.3. 3. Use case: Simple creation of several threads in a loop.

```
void foo()
{
   for (int i=0; i<NUM_THREADS; ++i)
      create_thread(&bar);
}
```

4.4.4. 4. Use case: Creation of several threads in a loop which are later joined.

```
void foo()
{
   for (int i=0; i<NUM_THREADS; ++i)
      threads[i] = create_thread(&bar);
   for (int i=0; i<NUM_THREADS; ++i)
      threads[i].join();
}
```

4.4.5. 5. Use case: Creation of a thread whose ownership is passed to another object/method.

```
void foo()
{
   thread = create_thread(&bar);
   manager.owns(thread);
}
```

4.4.6. 6. Use case: Creation of a thread whose ownership is shared between multiple objects.

```
void foo()
{
   thread = create_thread(&bar);
   manager1.add(thread);
   manager2.add(thread);
}
```

Of these usage patterns there's only one that requires reference management (number 6). Hopefully it's fairly obvious that this usage pattern simply won't occur as often as the other usage patterns. So there really isn't a "typical need" for a thread concept, though there is some need.

Since the need isn't typical we must use different criteria for deciding on either a thread_ref or thread design. Possible criteria include ease of use and performance. So let's analyze both of these carefully.

With ease of use we can look at existing experience. The standard C++ objects that represent a system resource, such as std::iostream, are noncopyable, so we know that C++ programmers must at least be experienced with this design. Most C++ developers are also used to smart pointers such as boost::shared_ptr, so we know they can at least adapt to a thread_ref concept with little effort. So existing experience isn't going to lead us to a choice.

The other thing we can look at is how difficult it is to use both types for the six usage patterns above. If we find it overly difficult to use a concept for any of the usage patterns there would be a good argument for choosing the other design. So we'll code all six usage patterns using both designs.

4.4.7. 1. Comparison: simple creation of a thread.

```
void foo()
{
   thread thrd(&bar);
}
void foo()
{
   thread_ref thrd = create_thread(&bar);
}
```

4.4.8. 2. Comparison: creation of a thread that's later joined.

```
void foo()
{
   thread thrd(&bar);
   thrd.join();
}
void foo()
{
   thread_ref thrd =
   create_thread(&bar);thrd->join();
}
```

4.4.9. 3. Comparison: simple creation of several threads in a loop.

```
void foo()
{
   for (int i=0; i<NUM_THREADS; ++i)
      thread thrd(&bar);
}
void foo()
{
   for (int i=0; i<NUM_THREADS; ++i)
      thread_ref thrd = create_thread(&bar);
}
```

4.4.10. 4. Comparison: creation of several threads in a loop which are later joined.

```
void foo()
{
   std::auto_ptr<thread> threads[NUM_THREADS];
   for (int i=0; i<NUM_THREADS; ++i)
      threads[i] = std::auto_ptr<thread>(new thread(&bar));
   for (int i= 0; i<NUM_THREADS;
       ++i)threads[i]->join();
}
void foo()
{
   thread_ref threads[NUM_THREADS];
   for (int i=0; i<NUM_THREADS; ++i)
      threads[i] = create_thread(&bar);
   for (int i= 0; i<NUM_THREADS;
      ++i)threads[i]->join();
}
```

4.4.11. 5. Comparison: creation of a thread whose ownership is passed to another object/method.

```
void foo()
{
   thread thrd* = new thread(&bar);
   manager.owns(thread);
}
void foo()
{
   thread_ref thrd = create_thread(&bar);
   manager.owns(thrd);
}
```

4.4.12. 6. Comparison: creation of a thread whose ownership is shared between multiple objects.

```
void foo()
{
   boost::shared_ptr<thread> thrd(new thread(&bar));
   manager1.add(thrd);
   manager2.add(thrd);
}
void foo()
{
   thread_ref thrd = create_thread(&bar);
   manager1.add(thrd);
   manager2.add(thrd);
}
```

This shows the usage patterns being nearly identical in complexity for both designs. The only actual added complexity occurs because of the use of operator new in (4), (5), and (6); and the use of std::auto_ptr and boost::shared_ptr in (4) and (6) respectively. However, that's not really much added complexity, and C++ programmers are used to using these idioms anyway. Some may dislike the presence of operator new in user code, but this can be eliminated by proper design of higher level concepts, such as the boost::thread_group class that simplifies example (4) down to:

```
void foo()
{
   thread_group threads;
   for (int i=0; i<NUM_THREADS; ++i)
      threads.create_thread(&bar);
   threads.join_all();
}
```

So ease of use is really a wash and not much help in picking a design.

So what about performance? Looking at the above code examples, we can analyze the theoretical impact to performance that both designs have. For (1) we can see that platforms that don't have a ref−counted native thread type (POSIX, for instance) will be impacted by a thread_ref design. Even if the native thread type is ref−counted there may be an impact if more state information has to be maintained for concepts foreign to the native API, such as clean up stacks for Win32 implementations. For (2) and (3) the performance impact will be identical to (1). For (4) things get a little more interesting and we find that theoretically at least the thread_ref may perform faster since the thread design requires dynamic memory allocation/deallocation. However, in practice there may be dynamic allocation for the thread_ref design as well, it will just be hidden from the user. As long as the implementation has to do dynamic allocations the thread_ref loses again because of the reference management. For (5) we see the same impact as we do for (4). For (6) we still have a possible impact to the thread design because of dynamic allocation but thread_ref no longer suffers because of its reference management, and in fact, theoretically at least, the thread_ref may do a better job of managing the references. All of this indicates that thread wins for (1), (2) and (3); with (4) and (5) the winner depending on the implementation and the platform but with the thread design probably having a better chance; and with (6) it will again depend on the implementation and platform but this time we favor thread_ref slightly. Given all of this it's a narrow margin, but the thread design prevails.

Given this analysis, and the fact that noncopyable objects for system resources are the normal designs that C++ programmers are used to dealing with, the **Boost.Threads** library has gone with a noncopyable design.

4.5. Rationale for not providing *Event Variables*

Event variables are simply far too error−prone. boost::condition variables are a much safer alternative. [Note that Graphical User Interface *events* are a different concept, and are not what is being discussed here.]

Event variables were one of the first synchronization primitives. They are still used today, for example, in the native Windows multithreading API. Yet both respected computer science researchers and experienced multithreading practitioners believe event variables are so inherently error−prone that they should never be used, and thus should not be part of a multithreading library.

Per Brinch Hansen [Hansen73] analyzed event variables in some detail, pointing out [emphasis his] that "*event operations force the programmer to be aware of the relative speeds of the sending and receiving processes*". His summary:

> We must therefore conclude that event variables of the previous type are impractical for system design. *The effect of an interaction between two processes must be independent of the speed at which it is carried out.*

Experienced programmers using the Windows platform today report that event variables are a continuing source of errors, even after previous bad experiences caused them to be very careful in their use of event variables. Overt problems can be avoided, for example, by teaming the event variable with a mutex, but that may just convert a race condition into another problem, such as excessive resource use. One of the most distressing aspects of the experience reports is the claim that many defects are latent. That is, the programs appear to work correctly, but contain hidden timing dependencies which will cause them to fail when environmental factors or usage patterns change, altering relative thread timings.

The decision to exclude event variables from **Boost.Threads** has been surprising to some Windows programmers. They have written programs which work using event variables, and wonder what the problem is. It seems similar to the "goto considered harmful" controversy of 30 years ago. It isn't that events, like gotos, can't be made to work, but rather that virtually all programs using alternatives will be easier to write, debug, read, maintain, and will be less likely to contain latent defects.

[Rationale provided by Beman Dawes]

5. Reference

5.1. Header <boost/thread/barrier.hpp>

```
namespace boost {
  class barrier;
}
```

Class barrier

boost::barrier

An object of class barrier is a synchronization primitive used to cause a set of threads to wait until they each perform a certain function or each reach a particular point in their execution.

Synopsis

```
class barrier : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  barrier(size_t);
  ~barrier();
  // waiting
  bool wait();
};
```

Description

When a barrier is created, it is initialized with a thread count N. The first N−1 calls to `wait()` will all cause their threads to be blocked. The Nth call to `wait()` will allow all of the waiting threads, including the Nth thread, to be placed in a ready state. The Nth call will also "reset" the barrier such that, if an additional N+1th call is made to `wait()`, it will be as though this were the first call to `wait()`; in other words, the N+1th to 2N−1th calls to `wait()` will cause their threads to be blocked, and the 2Nth call to `wait()` will allow all of the waiting threads, including the 2Nth thread, to be placed in a ready state and reset the barrier. This functionality allows the same set of N threads to re−use a barrier object to synchronize their execution at multiple points during their execution.

See Glossary for definitions of thread states blocked and ready. Note that "waiting" is a synonym for blocked.

`barrier` construct/copy/destruct

1. `barrier(size_t count);`

   Effects

   Constructs a barrier object that will cause `count` threads to block on a call to `wait()`.
2. `~barrier();`

   Effects

   Destroys `*this`. If threads are still executing their `wait()` operations, the behavior for these threads is undefined.

`barrier` waiting

1. `bool wait();`

   Effects

   Wait until N threads call `wait()`, where N equals the `count` provided to the constructor for the barrier object.

   Note that if the barrier is destroyed before `wait()` can return, the behavior is undefined.

   Returns

   Exactly one of the N threads will receive a return value of `true`, the others will receive a value of `false`. Precisely which thread receives the return value of `true` will be implementation−defined. Applications can use this value to designate one thread as a leader that will take a certain action, and the other threads emerging from the barrier can wait for that action to take place.

5.2. Header <boost/thread/condition.hpp>

```
namespace boost {
  class condition;
}
```

Class condition

boost::condition

An object of class condition is a synchronization primitive used to cause a thread to wait until a particular shared–data condition (or time) is met.

Synopsis

```
class condition : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  condition();
  ~condition();

  // notification
  void notify_one();
  void notify_all();

  // waiting
  template<typename ScopedLock> void wait(ScopedLock&);
  template<typename ScopedLock, typename Pred> void wait(ScopedLock&, Pred);
  template<typename ScopedLock>
    bool timed_wait(ScopedLock&, const boost::xtime&);
  template<typename ScopedLock, typename Pred>
    bool timed_wait(ScopedLock&, Pred);
};
```

Description

A condition object is always used in conjunction with a mutex object (an object whose type is a model of a Mutex or one of its refinements). The mutex object must be locked prior to waiting on the condition, which is verified by passing a lock object (an object whose type is a model of Lock or one of its refinements) to the condition object's wait functions. Upon blocking on the condition object, the thread unlocks the mutex object. When the thread returns from a call to one of the condition object's wait functions the mutex object is again locked. The tricky unlock/lock sequence is performed automatically by the condition object's wait functions.

The condition type is often used to implement the Monitor Object and other important patterns (see [SchmidtStalRohnertBuschmann] and [Hoare74]). Monitors are one of the most important patterns for creating reliable multithreaded programs.

See Glossary for definitions of thread states blocked and ready. Note that "waiting" is a synonym for blocked.

`condition` construct/copy/destruct

1. `condition();`

   Effects
      Constructs a condition object.
2. `~condition();`

   Effects
      Destroys `*this`.

`condition` notification

1. `void notify_one();`

   Effects
      If there is a thread waiting on `*this`, change that thread's state to ready. Otherwise there is no effect.

   Notes

If more than one thread is waiting on `*this`, it is unspecified which is made ready. After returning to a ready state the notified thread must still acquire the mutex again (which occurs within the call to one of the condition object's wait functions.)

2. `void notify_all();`

   Effects

   Change the state of all threads waiting on `*this` to ready. If there are no waiting threads, `notify_all()` has no effect.

`condition` waiting

1. `template<typename ScopedLock> void wait(ScopedLock& lock);`

   Requires

   `ScopedLock` meets the ScopedLock requirements.

   Effects

   Releases the lock on the mutex object associated with `lock`, blocks the current thread of execution until readied by a call to `this->notify_one()` or `this->notify_all()`, and then reacquires the lock.

   Throws

   lock_error if `!lock.locked()`

2.
   ```
   template<typename ScopedLock, typename Pred>
     void wait(ScopedLock& lock, Pred pred);
   ```

   Requires

   `ScopedLock` meets the ScopedLock requirements and the return from `pred()` is convertible to `bool`.

   Effects

   As if: `while (!pred()) wait(lock)`

   Throws

   lock_error if `!lock.locked()`

3.
   ```
   template<typename ScopedLock>
     bool timed_wait(ScopedLock& lock, const boost::xtime& xt);
   ```

   Requires

   `ScopedLock` meets the ScopedLock requirements.

   Effects

   Releases the lock on the mutex object associated with `lock`, blocks the current thread of execution until readied by a call to `this->notify_one()` or `this->notify_all()`, or until time `xt` is reached, and then reacquires the lock.

   Returns

   `false` if time `xt` is reached, otherwise `true`.

   Throws

   lock_error if `!lock.locked()`

4.
   ```
   template<typename ScopedLock, typename Pred>
     bool timed_wait(ScopedLock& lock, Pred pred);
   ```

   Requires

   `ScopedLock` meets the ScopedLock requirements and the return from `pred()` is convertible to `bool`.

   Effects

   As if: `while (!pred()) { if (!timed_wait(lock, xt)) return false; } return true;`

   Returns

   `false` if `xt` is reached, otherwise `true`.

   Throws

   lock_error if `!lock.locked()`

5.3. Header <boost/thread/exceptions.hpp>

```
namespace boost {
  class lock_error;
  class thread_resource_error;
}
```

Class lock_error

boost::lock_error

The lock_error class defines an exception type thrown to indicate a locking related error has been detected.

Synopsis

```
class lock_error : public std::logical_error {
public:
  // construct/copy/destruct
  lock_error();
};
```

Description

Examples of errors indicated by a lock_error exception include a lock operation which can be determined to result in a deadlock, or unlock operations attempted by a thread that does not own the lock.

`lock_error` construct/copy/destruct

1. `lock_error();`

   Effects
      Constructs a `lock_error` object.

Class thread_resource_error

boost::thread_resource_error

The thread_resource_error class defines an exception type that is thrown by constructors in the **Boost.Threads** library when thread–related resources can not be acquired.

Synopsis

```
class thread_resource_error : public std::runtime_error {
public:
  // construct/copy/destruct
  thread_resource_error();
};
```

Description

thread_resource_error is used only when thread–related resources cannot be acquired; memory allocation failures are indicated by std::bad_alloc.

`thread_resource_error` construct/copy/destruct

1. `thread_resource_error();`

   Effects

   Constructs a `thread_resource_error` object.

5.4. Header <boost/thread/mutex.hpp>

```
namespace boost {
  class mutex;
  class try_mutex;
  class timed_mutex;
}
```

Class mutex

boost::mutex

The mutex class is a model of the Mutex concept.

Synopsis

```
class mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;

  // construct/copy/destruct
  mutex();
  ~mutex();
};
```

Description

The mutex class is a model of the Mutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see try_mutex and timed_mutex.

For Recursive locking mechanics, see recursive_mutex, recursive_try_mutex, and recursive_timed_mutex.

The mutex class supplies the following typedef, which models the specified locking strategy:

Lock Name	Lock Concept
scoped_lock	ScopedLock

The mutex class uses an Unspecified locking strategy, so attempts to recursively lock a mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all mutex models in **Boost.Threads**, mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`mutex` construct/copy/destruct

1. `mutex();`

   Effects
      Constructs a mutex object.
   Postconditions
      `*this` is in an unlocked state.
2. `~mutex();`

   Effects
      Destroys a mutex object.
   Requires
      `*this` is in an unlocked state.
   Notes
      Danger: Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class try_mutex

boost::try_mutex

The try_mutex class is a model of the TryMutex concept.

Synopsis

```
class try_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;
  typedef implementation-defined scoped_try_lock;

  // construct/copy/destruct
  try_mutex();
  ~try_mutex();
};
```

Description

The try_mutex class is a model of the TryMutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see mutex and timed_mutex.

For Recursive locking mechanics, see recursive_mutex, recursive_try_mutex, and recursive_timed_mutex.

The try_mutex class supplies the following typedefs, which model the specified locking strategies:

Lock Name	Lock Concept
scoped_lock	ScopedLock
scoped_try_lock	ScopedTryLock

The try_mutex class uses an Unspecified locking strategy, so attempts to recursively lock a try_mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all mutex models in **Boost.Threads**, try_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`try_mutex` construct/copy/destruct

1. `try_mutex();`

   Effects
      Constructs a try_mutex object.
   Postconditions
      `*this` is in an unlocked state.
2. `~try_mutex();`

   Effects
      Destroys a try_mutex object.
   Requires
      `*this` is in an unlocked state.
   Notes
      Danger: Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class timed_mutex

boost::timed_mutex

The timed_mutex class is a model of the TimedMutex concept.

Synopsis

```
class timed_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;
  typedef implementation-defined scoped_try_lock;
  typedef implementation-defined scoped_timed_lock;

  // construct/copy/destruct
  timed_mutex();
  ~timed_mutex();
};
```

Description

The timed_mutex class is a model of the TimedMutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see mutex and try_mutex.

For Recursive locking mechanics, see recursive_mutex, recursive_try_mutex, and recursive_timed_mutex.

The timed_mutex class supplies the following typedefs, which model the specified locking strategies:

Lock Name	Lock Concept
scoped_lock	ScopedLock
scoped_try_lock	ScopedTryLock
scoped_timed_lock	ScopedTimedLock

The timed_mutex class uses an Unspecified locking strategy, so attempts to recursively lock a timed_mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all mutex models in **Boost.Threads**, timed_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`timed_mutex` construct/copy/destruct

1. `timed_mutex();`

   Effects
   : Constructs a timed_mutex object.

   Postconditions
   : `*this` is in an unlocked state.
2. `~timed_mutex();`

   Effects
   : Destroys a timed_mutex object.

   Requires
   : `*this` is in an unlocked state.

   Notes
   : **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

5.5. Header <boost/thread/once.hpp>

```
BOOST_ONCE_INIT

namespace boost {
  typedef implementation-defined once_flag;  // The call_once function and
once_flag type (statically initialized to
BOOST_ONCE_INIT) can be used to run a
                             routine exactly once. This can be used to initialize data in a
thread-safe
                             manner.
   call_once(void (*func)(), once_flag&);
}
```

Macro BOOST_ONCE_INIT

BOOST_ONCE_INIT   The call_once function and `once_flag` type (statically initialized to BOOST_ONCE_INIT) can be used to run a routine exactly once. This can be used to initialize data in a thread–safe manner.

Synopsis

```
BOOST_ONCE_INIT
```

Description

The implementation−defined macro BOOST_ONCE_INIT is a constant value used to initialize `once_flag` instances to indicate that the logically associated routine has not been run yet. See call_once for more details.

Function call_once

boost::call_once   The call_once function and `once_flag` type (statically initialized to BOOST_ONCE_INIT) can be used to run a routine exactly once. This can be used to initialize data in a thread−safe manner.

Synopsis

```
 call_once(void (*func)() func, once_flag& flag);
```

Description

Example usage is as follows:

```
//Example usage:
boost::once_flag once = BOOST_ONCE_INIT;

void init()
{
    //...
}

void thread_proc()
{
    boost::call_once(&init, once);
}
```

Requires
: The function `func` shall not throw exceptions.

Effects
: As if (in an atomic fashion): `if (flag == BOOST_ONCE_INIT) func();`

Postconditions
: `flag != BOOST_ONCE_INIT`

5.6. Header <boost/thread/recursive_mutex.hpp>

```
namespace boost {
  class recursive_mutex;
  class recursive_try_mutex;
  class recursive_timed_mutex;
}
```

Class recursive_mutex

boost::recursive_mutex

The recursive_mutex class is a model of the Mutex concept.

Synopsis

```
class recursive_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;

  // construct/copy/destruct
  recursive_mutex();
  ~recursive_mutex();
};
```

Description

The recursive_mutex class is a model of the Mutex concept. It should be used to synchronize access to shared resources using Recursive locking mechanics.

For classes that model related mutex concepts, see recursive_try_mutex and recursive_timed_mutex.

For Unspecified locking mechanics, see mutex, try_mutex, and timed_mutex.

The recursive_mutex class supplies the following typedef, which models the specified locking strategy:

Table 9.23. Supported Lock Types

Lock Name	Lock Concept
scoped_lock	ScopedLock

The recursive_mutex class uses a Recursive locking strategy, so attempts to recursively lock a recursive_mutex object succeed and an internal "lock count" is maintained. Attempts to unlock a recursive_mutex object by threads that don't own a lock on it result in **undefined behavior**.

Like all mutex models in **Boost.Threads**, recursive_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`recursive_mutex` construct/copy/destruct

1. `recursive_mutex();`

   Effects
      Constructs a recursive_mutex object.
   Postconditions
      `*this` is in an unlocked state.
2. `~recursive_mutex();`

   Effects
      Destroys a recursive_mutex object.
   Requires
      `*this` is in an unlocked state.
   Notes
      Danger: Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class recursive_try_mutex

boost::recursive_try_mutex

The recursive_try_mutex class is a model of the TryMutex concept.

Synopsis

```
class recursive_try_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;
  typedef implementation-defined scoped_try_lock;

  // construct/copy/destruct
  recursive_try_mutex();
  ~recursive_try_mutex();
};
```

Description

The recursive_try_mutex class is a model of the TryMutex concept. It should be used to synchronize access to shared resources using Recursive locking mechanics.

For classes that model related mutex concepts, see recursive_mutex and recursive_timed_mutex.

For Unspecified locking mechanics, see mutex, try_mutex, and timed_mutex.

The recursive_try_mutex class supplies the following typedefs, which model the specified locking strategies:

Table 9.24. Supported Lock Types

Lock Name	Lock Concept
scoped_lock	ScopedLock
scoped_try_lock	ScopedTryLock

The recursive_try_mutex class uses a Recursive locking strategy, so attempts to recursively lock a recursive_try_mutex object succeed and an internal "lock count" is maintained. Attempts to unlock a recursive_mutex object by threads that don't own a lock on it result in **undefined behavior**.

Like all mutex models in **Boost.Threads**, recursive_try_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`recursive_try_mutex` construct/copy/destruct

1. `recursive_try_mutex();`

   Effects
      Constructs a recursive_try_mutex object.
   Postconditions
      `*this` is in an unlocked state.
2. `~recursive_try_mutex();`

Effects
: Destroys a recursive_try_mutex object.

Requires
: `*this` is in an unlocked state.

Notes
: **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class recursive_timed_mutex

boost::recursive_timed_mutex

The recursive_timed_mutex class is a model of the TimedMutex concept.

Synopsis

```
class recursive_timed_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;
  typedef implementation-defined scoped_try_lock;
  typedef implementation-defined scoped_timed_lock;

  // construct/copy/destruct
  recursive_timed_mutex();
  ~recursive_timed_mutex();
};
```

Description

The recursive_timed_mutex class is a model of the TimedMutex concept. It should be used to synchronize access to shared resources using Recursive locking mechanics.

For classes that model related mutex concepts, see recursive_mutex and recursive_try_mutex.

For Unspecified locking mechanics, see mutex, try_mutex, and timed_mutex.

The recursive_timed_mutex class supplies the following typedefs, which model the specified locking strategies:

Table 9.25. Supported Lock Types

Lock Name	Lock Concept
scoped_lock	ScopedLock
scoped_try_lock	ScopedTryLock
scoped_timed_lock	ScopedTimedLock

The recursive_timed_mutex class uses a Recursive locking strategy, so attempts to recursively lock a recursive_timed_mutex object succeed and an internal "lock count" is maintained. Attempts to unlock a recursive_mutex object by threads that don't own a lock on it result in **undefined behavior**.

Like all mutex models in **Boost.Threads**, recursive_timed_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`recursive_timed_mutex` construct/copy/destruct

1. `recursive_timed_mutex();`

   Effects
   : Constructs a recursive_timed_mutex object.

   Postconditions
   : `*this` is in an unlocked state.

2. `~recursive_timed_mutex();`

   Effects
   : Destroys a recursive_timed_mutex object.

   Requires
   : `*this` is in an unlocked state.

   Notes
   : **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

5.7. Header <boost/thread/read_write_mutex.hpp>

```
namespace boost {
  class read_write_mutex;
  class try_read_write_mutex;
  class timed_read_write_mutex;
  namespace read_write_scheduling_policy {
    enum read_write_scheduling_policy;
  }
}
```

Type read_write_scheduling_policy

boost::read_write_scheduling_policy::read_write_scheduling_policy

Specifies the inter−class sheduling policy to use when a set of threads try to obtain different types of locks simultaneously.

Synopsis

```
enum read_write_scheduling_policy { writer_priority, reader_priority,
                                    alternating_many_reads,
                                    alternating_single_read };
```

Class read_write_mutex

boost::read_write_mutex

The read_write_mutex class is a model of the ReadWriteMutex concept.

Synopsis

```
class read_write_mutex  : private boost::noncopyable,   // Exposition only
                          private boost::noncopyable    // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_read_write_lock;
  typedef implementation-defined scoped_read_lock;
  typedef implementation-defined scoped_write_lock;

  // construct/copy/destruct
  read_write_mutex(boost::read_write_scheduling_policy);
  ~read_write_mutex();
};
```

Description

The read_write_mutex class is a model of the ReadWriteMutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see try_read_write_mutex and timed_read_write_mutex.

The read_write_mutex class supplies the following typedefs, which model the specified locking strategies:

Lock Name	**Lock Concept**
scoped_read_write_lock	ScopedReadWriteLock
scoped_read_lock	ScopedLock
scoped_write_lock	ScopedLock

The read_write_mutex class uses an Unspecified locking strategy, so attempts to recursively lock a read_write_mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all read/write mutex models in **Boost.Threads**, read_write_mutex has two types of scheduling policies, an inter–class sheduling policy between threads trying to obtain different types of locks and an intra–class sheduling policy between threads trying to obtain the same type of lock. The read_write_mutex class allows the programmer to choose what inter–class sheduling policy will be used; however, like all read/write mutex models, read_write_mutex leaves the intra–class sheduling policy as Unspecified.

Note

Self–deadlock is virtually guaranteed if a thread tries to lock the same read_write_mutex multiple times unless all locks are read–locks (but see below)

`read_write_mutex` construct/copy/destruct

1. `read_write_mutex(boost::read_write_scheduling_policy count);`

   Effects
      Constructs a read_write_mutex object.
   Postconditions
      `*this` is in an unlocked state.
2. `~read_write_mutex();`

   Effects
      Destroys a read_write_mutex object.
   Requires
      `*this` is in an unlocked state.
   Notes
      Danger: Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class try_read_write_mutex

boost::try_read_write_mutex

The try_read_write_mutex class is a model of the TryReadWriteMutex concept.

Synopsis

```
class try_read_write_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_read_write_lock;
  typedef implementation-defined scoped_try_read_write_lock;
  typedef implementation-defined scoped_read_lock;
  typedef implementation-defined scoped_try_read_lock;
  typedef implementation-defined scoped_write_lock;
  typedef implementation-defined scoped_try_write_lock;

  // construct/copy/destruct
  try_read_write_mutex(boost::read_write_scheduling_policy);
  ~try_read_write_mutex();
};
```

Description

The try_read_write_mutex class is a model of the TryReadWriteMutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see read_write_mutex and timed_read_write_mutex.

The try_read_write_mutex class supplies the following typedefs, which model the specified locking strategies:

Lock Name	Lock Concept
scoped_read_write_lock	ScopedReadWriteLock
scoped_try_read_write_lock	ScopedTryReadWriteLock
scoped_read_lock	ScopedLock
scoped_try_read_lock	ScopedTryLock
scoped_write_lock	ScopedLock
scoped_try_write_lock	ScopedTryLock

The try_read_write_mutex class uses an Unspecified locking strategy, so attempts to recursively lock a try_read_write_mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all read/write mutex models in **Boost.Threads**, try_read_write_mutex has two types of scheduling policies, an inter−class sheduling policy between threads trying to obtain different types of locks and an intra−class sheduling policy between threads trying to obtain the same type of lock. The try_read_write_mutex class allows the programmer to choose what inter−class sheduling policy will be used; however, like all read/write mutex models, try_read_write_mutex leaves the intra−class sheduling policy as Unspecified.

Note

Self−deadlock is virtually guaranteed if a thread tries to lock the same try_read_write_mutex multiple times unless all locks are read−locks (but see below)

`try_read_write_mutex` construct/copy/destruct

1. `try_read_write_mutex(boost::read_write_scheduling_policy count);`

   Effects
   : Constructs a try_read_write_mutex object.

   Postconditions
   : `*this` is in an unlocked state.

2. `~try_read_write_mutex();`

   Effects
   : Destroys a try_read_write_mutex object.

   Requires
   : `*this` is in an unlocked state.

   Notes
   : **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class timed_read_write_mutex

boost::timed_read_write_mutex

The timed_read_write_mutex class is a model of the TimedReadWriteMutex concept.

Synopsis

```
class timed_read_write_mutex {
public:
  // types
  typedef implementation-defined scoped_read_write_lock;
  typedef implementation-defined scoped_try_read_write_lock;
  typedef implementation-defined scoped_timed_read_write_lock;
  typedef implementation-defined scoped_read_lock;
  typedef implementation-defined scoped_try_read_lock;
  typedef implementation-defined scoped_timed_read_lock;
  typedef implementation-defined scoped_write_lock;
  typedef implementation-defined scoped_try_write_lock;
  typedef implementation-defined scoped_timed_write_lock;

  // construct/copy/destruct
  timed_read_write_mutex(boost::read_write_scheduling_policy);
  ~timed_read_write_mutex();
};
```

Description

The timed_read_write_mutex class is a model of the TimedReadWriteMutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see read_write_mutex and try_read_write_mutex.

The timed_read_write_mutex class supplies the following typedefs, which model the specified locking strategies:

Lock Name	Lock Concept
scoped_read_write_lock	ScopedReadWriteLock
scoped_try_read_write_lock	ScopedTryReadWriteLock
scoped_timed_read_write_lock	ScopedTimedReadWriteLock
scoped_read_lock	ScopedLock
scoped_try_read_lock	ScopedTryLock
scoped_timed_read_lock	ScopedTimedLock
scoped_write_lock	ScopedLock
scoped_try_write_lock	ScopedTryLock
scoped_timed_write_lock	ScopedTimedLock

The timed_read_write_mutex class uses an Unspecified locking strategy, so attempts to recursively lock a timed_read_write_mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all read/write mutex models in **Boost.Threads**, timed_read_write_mutex has two types of scheduling policies, an inter–class sheduling policy between threads trying to obtain different types of locks and an intra–class sheduling policy between threads trying to obtain the same type of lock. The timed_read_write_mutex class allows the programmer to choose what inter–class sheduling policy will be used; however, like all read/write mutex models, timed_read_write_mutex leaves the intra–class sheduling policy as Unspecified.

Note

Self–deadlock is virtually guaranteed if a thread tries to lock the same timed_read_write_mutex multiple times unless all locks are read–locks (but see below)

`timed_read_write_mutex` construct/copy/destruct

1. `timed_read_write_mutex(boost::read_write_scheduling_policy count);`

   Effects
   : Constructs a timed_read_write_mutex object.

   Postconditions
   : `*this` is in an unlocked state.

2. `~timed_read_write_mutex();`

   Effects
   : Destroys a timed_read_write_mutex object.

   Requires
   : `*this` is in an unlocked state.

   Notes
   : **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

5.8. Header <boost/thread/thread.hpp>

```
namespace boost {
  class thread;
  class thread_group;
}
```

Class thread

boost::thread

The thread class represents threads of execution, and provides the functionality to create and manage threads within the **Boost.Threads** library. See Glossary for a precise description of thread of execution, and for definitions of threading−related terms and of thread states such as blocked.

Synopsis

```
class thread : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  thread();
  explicit thread(const boost::function0<void>&);
  ~thread();

  // comparison
  bool operator==() const;
  bool operator!=() const;

  // modifier
  void join();

  // static
  static void sleep(const xtime&);
  static void yield();
};
```

Description

A thread of execution has an initial function. For the program's initial thread, the initial function is `main()`. For other threads, the initial function is `operator()` of the function object passed to the thread object's constructor.

A thread of execution is said to be "finished" or to have "finished execution" when its initial function returns or is terminated. This includes completion of all thread cleanup handlers, and completion of the normal C++ function return behaviors, such as destruction of automatic storage (stack) objects and releasing any associated implementation resources.

A thread object has an associated state which is either "joinable" or "non−joinable".

Except as described below, the policy used by an implementation of **Boost.Threads** to schedule transitions between thread states is unspecified.

Note

Just as the lifetime of a file may be different from the lifetime of an `iostream` object which represents the file, the lifetime of a thread of execution may be different from the thread object which represents the thread of execution. In particular, after a call to `join()`, the thread of execution will no longer exist even though the thread object continues to exist until the end of its normal lifetime. The converse is also possible; if a thread object is destroyed without `join()` first having been called, the thread of execution continues until its initial function completes.

`thread` construct/copy/destruct

1. `thread();`

Effects
: Constructs a thread object representing the current thread of execution.

Postconditions
: `*this` is non–joinable.

Notes
: **Danger:** `*this` is valid only within the current thread.

2. `explicit thread(const boost::function0<void>& threadfunc);`

Effects
: Starts a new thread of execution and constructs a thread object representing it. Copies `threadfunc` (which in turn copies the function object wrapped by `threadfunc`) to an internal location which persists for the lifetime of the new thread of execution. Calls `operator()` on the copy of the `threadfunc` function object in the new thread of execution.

Postconditions
: `*this` is joinable.

Throws
: `boost::thread_resource_error` if a new thread of execution cannot be started.

3. `~thread();`

Effects
: Destroys `*this`. The actual thread of execution may continue to execute after the thread object has been destroyed.

Notes
: If `*this` is joinable the actual thread of execution becomes "detached". Any resources used by the thread will be reclaimed when the thread of execution completes. To ensure such a thread of execution runs to completion before the thread object is destroyed, call `join()`.

`thread` comparison

1. `bool operator==( rhs) const;`

Requires
: The thread is non–terminated or `*this` is joinable.

Returns
: `true` if `*this` and `rhs` represent the same thread of execution.

2. `bool operator!=( rhs) const;`

Requires
: The thread is non–terminated or `*this` is joinable.

Returns
: `!(*this==rhs)`.

`thread` modifier

1. `void join();`

Requires
: `*this` is joinable.

Effects
: The current thread of execution blocks until the initial function of the thread of execution represented by `*this` finishes and all resources are reclaimed.

Notes
: If `*this == thread()` the result is implementation–defined. If the implementation doesn't detect this the result will be deadlock.

`thread` static

1. `static void sleep(const xtime& xt);`

   Effects
   : The current thread of execution blocks until `xt` is reached.

2. `static void yield();`

   Effects
   : The current thread of execution is placed in the ready state.

   Notes
   : Allow the current thread to give up the rest of its time slice (or other scheduling quota) to another thread. Particularly useful in non–preemptive implementations.

Class thread_group

boost::thread_group The thread_group class provides a container for easy grouping of threads to simplify several common thread creation and management idioms.

Synopsis

```
class thread_group : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  thread_group();
  ~thread_group();

  // modifier
  thread* create_thread(const boost::function0<void>&);
  void add_thread(thread* thrd);
  void remove_thread(thread* thrd);
  void join_all();
};
```

Description

`thread_group` construct/copy/destruct

1. `thread_group();`

   Effects
   : Constructs an empty thread_group container.

2. `~thread_group();`

   Effects
   : Destroys each contained thread object. Destroys `*this`.

   Notes
   : Behavior is undefined if another thread references `*this` during the execution of the destructor.

`thread_group` modifier

1. `thread* create_thread(const boost::function0<void>& threadfunc);`

   Effects

Creates a new thread object that executes `threadfunc` and adds it to the `thread_group` container object's list of managed thread objects.

Returns

Pointer to the newly created thread object.

2. `void add_thread(thread* thrd thrd);`

   Effects

   Adds `thrd` to the thread_group object's list of managed thread objects. The `thrd` object must have been allocated via `operator new` and will be deleted when the group is destroyed.

3. `void remove_thread(thread* thrd thrd);`

   Effects

   Removes `thread` from `*this`'s list of managed thread objects.

   Throws

   ??? if `thrd` is not in `*this`'s list of managed thread objects.

4. `void join_all();`

   Effects

   Calls `join()` on each of the managed thread objects.

5.9. Header <boost/thread/tss.hpp>

```
namespace boost {
  class thread_specific_ptr;
}
```

Class thread_specific_ptr

boost::thread_specific_ptr   The thread_specific_ptr class defines an interface for using thread specific storage.

Synopsis

```
class thread_specific_ptr : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  thread_specific_ptr();
  thread_specific_ptr(void (*cleanup)(void*));
  ~thread_specific_ptr();

  // modifier functions
  T* release();
  void reset(T* = 0);

  // observer functions
  T* get() const;
  T* operator->() const;
  T& operator*()() const;
};
```

Description

Thread specific storage is data associated with individual threads and is often used to make operations that rely on global data thread−safe.

Template thread_specific_ptr stores a pointer to an object obtained on a thread−by−thread basis and calls a specified cleanup handler on the contained pointer when the thread terminates. The cleanup handlers are called in the reverse order of construction of the thread_specific_ptrs, and for the initial thread are called by the destructor, providing the same ordering guarantees as for normal declarations. Each thread initially stores the null pointer in each thread_specific_ptr instance.

The template thread_specific_ptr is useful in the following cases:

- An interface was originally written assuming a single thread of control and it is being ported to a multithreaded environment.
- Each thread of control invokes sequences of methods that share data that are physically unique for each thread, but must be logically accessed through a globally visible access point instead of being explicitly passed.

`thread_specific_ptr` construct/copy/destruct

1. `thread_specific_ptr();`

   Requires
      The expression `delete get()` is well formed.

   Effects
      A thread−specific data key is allocated and visible to all threads in the process. Upon creation, the value `NULL` will be associated with the new key in all active threads. A cleanup method is registered with the key that will call `delete` on the value associated with the key for a thread when it exits. When a thread exits, if a key has a registered cleanup method and the thread has a non−`NULL` value associated with that key, the value of the key is set to `NULL` and then the cleanup method is called with the previously associated value as its sole argument. The order in which registered cleanup methods are called when a thread exits is undefined. If after all the cleanup methods have been called for all non−`NULL` values, there are still some non−`NULL` values with associated cleanup handlers the result is undefined behavior.

   Throws
      boost::thread_resource_error if the necessary resources can not be obtained.

   Notes
      There may be an implementation specific limit to the number of thread specific storage objects that can be created, and this limit may be small.

   Rationale
      The most common need for cleanup will be to call `delete` on the associated value. If other forms of cleanup are required the overloaded constructor should be called instead.

2. `thread_specific_ptr(`**`void`**` (*cleanup)(`**`void`**`*) cleanup);`

   Effects
      A thread−specific data key is allocated and visible to all threads in the process. Upon creation, the value `NULL` will be associated with the new key in all active threads. The `cleanup` method is registered with the key and will be called for a thread with the value associated with the key for that thread when it exits. When a thread exits, if a key has a registered cleanup method and the thread has a non−`NULL` value associated with that key, the value of the key is set to `NULL` and then the cleanup method is called with the previously associated value as its sole argument. The order in which registered cleanup methods are called when a thread exits is undefined. If after all the cleanup methods have been called for all non−`NULL` values, there are still some non−`NULL` values with associated cleanup handlers the result is undefined behavior.

   Throws
      boost::thread_resource_error if the necessary resources can not be obtained.

   Notes
      There may be an implementation specific limit to the number of thread specific storage objects that can be created, and this limit may be small.

   Rationale
      There is the occasional need to register specialized cleanup methods, or to register no cleanup method at all (done by passing `NULL` to this constructor.

3. `~thread_specific_ptr();`

   Effects

Deletes the thread–specific data key allocated by the constructor. The thread–specific data values associated with the key need not be `NULL`. It is the responsibility of the application to perform any cleanup actions for data associated with the key.

Notes

Does not destroy any data that may be stored in any thread's thread specific storage. For this reason you should not destroy a thread_specific_ptr object until you are certain there are no threads running that have made use of its thread specific storage.

Rationale

Associated data is not cleaned up because registered cleanup methods need to be run in the thread that allocated the associated data to be guarranteed to work correctly. There's no safe way to inject the call into another thread's execution path, making it impossible to call the cleanup methods safely.

`thread_specific_ptr` modifier functions

1. `T* release();`

   Postconditions

   `*this` holds the null pointer for the current thread.

   Returns

   `this->get()` prior to the call.

   Rationale

   This method provides a mechanism for the user to relinquish control of the data associated with the thread–specific key.

2. `void reset(T* p = 0);`

   Effects

   If `this->get() != p && this->get() != NULL` then call the associated cleanup function.

   Postconditions

   `*this` holds the pointer `p` for the current thread.

`thread_specific_ptr` observer functions

1. `T* get() const;`

   Returns

   The object stored in thread specific storage for the current thread for `*this`.

   Notes

   Each thread initially returns 0.

2. `T* operator->() const;`

   Returns

   `this->get()`.

3. `T& operator*()() const;`

   Requires

   `this->get() != 0`

   Returns

   `this->get()`.

5.10. Header <boost/thread/xtime.hpp>

```
namespace boost {
  enum xtime_clock_types;

  struct xtime;
  int xtime_get(xtime*, int);
}
```

Type xtime_clock_types

boost::xtime_clock_types

Specifies the clock type to use when creating an object of type xtime.

Synopsis

```
enum xtime_clock_types { TIME_UTC };
```

Struct xtime

boost::xtime

An object of type xtime defines a time that is used to perform high–resolution time operations. This is a temporary solution that will be replaced by a more robust time library once available in Boost.

Synopsis

```
struct xtime {

  platform-specific-type sec;
};

// creation
int xtime_get(xtime*, int);
```

Description

The xtime type is used to represent a point on some time scale or a duration in time. This type may be proposed for the C standard by Markus Kuhn. **Boost.Threads** provides only a very minimal implementation of this proposal; it is expected that a full implementation (or some other time library) will be provided in Boost as a separate library, at which time **Boost.Threads** will deprecate its own implementation.

Note that the resolution is implementation specific. For many implementations the best resolution of time is far more than one nanosecond, and even when the resolution is reasonably good, the latency of a call to `xtime_get()` may be significant. For maximum portability, avoid durations of less than one second.

`xtime` creation

1. `int xtime_get(xtime* xtp, int clock_type);`

   Postconditions

   `xtp` represents the current point in time as a duration since the epoch specified by `clock_type`.

   Returns

   `clock_type` if successful, otherwise 0.

6. Frequently Asked Questions

6.1. Are lock objects thread safe?

No! Lock objects are not meant to be shared between threads. They are meant to be short−lived objects created on automatic storage within a code block. Any other usage is just likely to lead to errors and won't really be of actual benefit anyway. Share Mutexes, not Locks. For more information see the rationale behind the design for lock objects.

6.2. Why was **Boost.Threads** modeled after (specific library name)?

It wasn't. **Boost.Threads** was designed from scratch. Extensive design discussions involved numerous people representing a wide range of experience across many platforms. To ensure portability, the initial implements were done in parallel using POSIX Threads and the Win32 threading API. But the **Boost.Threads** design is very much in the spirit of C++, and thus doesn't model such C based APIs.

6.3. Why wasn't **Boost.Threads** modeled after (specific library name)?

Existing C++ libraries either seemed dangerous (often failing to take advantage of prior art to reduce errors) or had excessive dependencies on library components unrelated to threading. Existing C libraries couldn't meet our C++ requirements, and were also missing certain features. For instance, the WIN32 thread API lacks condition variables, even though these are critical for the important Monitor pattern [SchmidtStalRohnertBuschmann].

6.4. Why do Mutexes have noncopyable semantics?

To ensure that deadlocks don't occur. The only logical form of copy would be to use some sort of shallow copy semantics in which multiple mutex objects could refer to the same mutex state. This means that if ObjA has a mutex object as part of its state and ObjB is copy constructed from it, then when ObjB::foo() locks the mutex it has effectively locked ObjA as well. This behavior can result in deadlock. Other copy semantics result in similar problems (if you think you can prove this to be wrong then supply us with an alternative and we'll reconsider).

6.5. How can you prevent deadlock from occurring when a thread must lock multiple mutexes?

Always lock them in the same order. One easy way of doing this is to use each mutex's address to determine the order in which they are locked. A future **Boost.Threads** concept may wrap this pattern up in a reusable class.

6.6. Don't noncopyable Mutex semantics mean that a class with a mutex member will be noncopyable as well?

No, but what it does mean is that the compiler can't generate a copy constructor and assignment operator, so they will have to be coded explicitly. This is a **good thing**, however, since the compiler generated operations would not be thread−safe. The following is a simple example of a class with copyable semantics and internal synchronization through a mutex member.

```
class counter
{
public:
   // Doesn't need synchronization since there can be no references to *this
   // until after it's constructed!
   explicit counter(int initial_value)
      : m_value(initial_value)
   {
   }
   // We only need to synchronize other for the same reason we don't have to
   // synchronize on construction!
   counter(const counter& other)
   {
      boost::mutex::scoped_lock scoped_lock(other.m_mutex);
      m_value = other.m_value;
   }
   // For assignment we need to synchronize both objects!
   const counter& operator=(const counter& other)
   {
      if (this == &other)
         return *this;
      boost::mutex::scoped_lock lock1(&m_mutex < &other.m_mutex ? m_mutex : other.m_mutex);
      boost::mutex::scoped_lock lock2(&m_mutex > &other.m_mutex ? m_mutex : other.m_mutex);
```

```
         m_value = other.m_value;
         return *this;
      }
      int value() const
      {
         boost::mutex::scoped_lock scoped_lock(m_mutex);
         return m_value;
      }
      int increment()
      {
         boost::mutex::scoped_lock scoped_lock(m_mutex);
         return ++m_value;
      }
   private:
      mutable boost::mutex m_mutex;
      int m_value;
   };
```

6.7. How can you lock a Mutex member in a const member function, in order to implement the Monitor Pattern?

The Monitor Pattern [SchmidtStalRohnertBuschmann] mutex should simply be declared as mutable. See the example code above. The internal state of mutex types could have been made mutable, with all lock calls made via const functions, but this does a poor job of documenting the actual semantics (and in fact would be incorrect since the logical state of a locked mutex clearly differs from the logical state of an unlocked mutex). Declaring a mutex member as mutable clearly documents the intended semantics.

6.8. Why supply boost::condition variables rather than event variables?

Condition variables result in user code much less prone to race conditions than event variables. See Section 4.5, Rationale for not providing *Event Variables* for analysis. Also see [Hoare74] and [SchmidtStalRohnertBuschmann].

6.9. Why isn't thread cancellation or termination provided?

There's a valid need for thread termination, so at some point **Boost.Threads** probably will include it, but only after we can find a truly safe (and portable) mechanism for this concept.

6.10. Is it safe for threads to share automatic storage duration (stack) objects via pointers or references?

Only if you can guarantee that the lifetime of the stack object will not end while other threads might still access the object. Thus the safest practice is to avoid sharing stack objects, particularly in designs where threads are created and destroyed dynamically. Restrict sharing of stack objects to simple designs with very clear and unchanging function and thread lifetimes. (Suggested by Darryl Green).

6.11. Why has class semaphore disappeared?

Semaphore was removed as too error prone. The same effect can be achieved with greater safety by the combination of a mutex and a condition variable.

7. Configuration

Boost.Threads uses several configuration macros in <boost/config.hpp>, as well as configuration macros meant to be supplied by the application. These macros are documented here.

7.1. Library Defined Public Macros

These macros are defined by **Boost.Threads** but are expected to be used by application code.

Macro	Meaning
BOOST_HAS_THREADS	Indicates that threading support is available. This means both that there is a platform specific implementation for **Boost.Threads** and that threading support has been enabled in a platform specific manner. For instance, on the Win32 platform there's an implementation for **Boost.Threads** but unless the program is compiled against one of the multithreading runtimes (often determined by the compiler predefining the macro _MT) the BOOST_HAS_THREADS macro remains undefined.

7.2. Library Defined Implementation Macros

These macros are defined by **Boost.Threads** and are implementation details of interest only to implementors.

Macro	Meaning
BOOST_HAS_WINTHREADS	Indicates that the platform has the Microsoft Win32 threading libraries, and that they should be used to implement **Boost.Threads**.
BOOST_HAS_PTHREADS	Indicates that the platform has the POSIX pthreads libraries, and that they should be used to implement **Boost.Threads**.
BOOST_HAS_FTIME	Indicates that the implementation should use GetSystemTimeAsFileTime() and the FILETIME type to calculate the current time. This is an implementation detail used by boost::detail::getcurtime().
BOOST_HAS_GETTTIMEOFDAY	Indicates that the implementation should use gettimeofday() to calculate the current time. This is an implementation detail used by boost::detail::getcurtime().

8. Build

How you build the **Boost.Threads** libraries, and how you build your own applications that use those libraries, are some of the most frequently asked questions. Build processes are difficult to deal with in a portable manner. That's one reason why **Boost.Threads** makes use of **Boost.Build**. In general you should refer to the documentation for **Boost.Build**. This document will only supply you with some simple usage examples for how to use *bjam* to build and test **Boost.Threads**. In addition, this document will try to explain the build requirements so that users may create their own build processes (for instance, create an IDE specific project), both for building and testing **Boost.Threads**, as well as for building their own projects using **Boost.Threads**.

8.1. Building the Boost.Threads Libraries

To build the **Boost.Threads** libraries using **Boost.Build**, simply change to the directory *boost_root*/libs/thread/build and execute the command:

```
bjam −sTOOLS=toolset
```

This will create the debug and the release builds of the **Boost.Threads** library.

Note

Invoking the above command in *boost_root* will build all of the Boost distribution, including **Boost.Threads**.
The Jamfile supplied with **Boost.Threads** produces a dynamic link library named *boost_thread{build−specific−tags}.{extension}*, where the build−specific tags indicate the toolset used to build the library, whether it's a debug or release build, what version of Boost was used, etc.; and the extension is the appropriate extension for a dynamic link library for the platform for which **Boost.Threads** is being built. For instance, a debug library built for Win32 with VC++ 7.1 using Boost 1.31 would be named *boost_thread−vc71−mt−gd−1_31.dll*.

The source files that are used to create the **Boost.Threads** library are all of the *.cpp files found in *boost_root*/libs/thread/src. These need to be built with the compiler's and linker's multi−threading support enabled. If you want to create your own build solution you'll have to follow these same guidelines. One of the most frequently reported problems when trying to do this occurs from not enabling the compiler's and linker's support for multi−threading.

8.2. Testing the Boost.Threads Libraries

To test the **Boost.Threads** libraries using **Boost.Build**, simply change to the directory *boost_root*/libs/thread/test and execute the command:

```
bjam -sTOOLS=toolset test
```

9. Implementation Notes

9.1. Win32

In the current Win32 implementation, creating a boost::thread object during dll initialization will result in deadlock because the thread class constructor causes the current thread to wait on the thread that is being created until it signals that it has finished its initialization, and, as stated in the MSDN Library, "DllMain" article, "Remarks" section, "Because DLL notifications are serialized, entry−point functions should not attempt to communicate with other threads or processes. Deadlocks may occur as a result." (Also see "Under the Hood", January 1996 for a more detailed discussion of this issue).

The following non−exhaustive list details some of the situations that should be avoided until this issue can be addressed:

- Creating a boost::thread object in DllMain() or in any function called by it.
- Creating a boost::thread object in the constructor of a global static object or in any function called by one.
- Creating a boost::thread object in MFC's CWinApp::InitInstance() function or in any function called by it.
- Creating a boost::thread object in the function pointed to by MFC's _pRawDllMain function pointer or in any function called by it.

10. Release Notes

10.1. Boost 1.32.0

10.1.1. Documentation converted to BoostBook

The documentation was converted to BoostBook format, and a number of errors and inconsistencies were fixed in the process. Since this was a fairly large task, there are likely to be more errors and inconsistencies remaining. If you find any, please report them!

10.1.2. Statically−link build option added

The option to link **Boost.Threads** as a static library has been added (with some limitations on Win32 platforms). This feature was originally removed from an earlier version of Boost because boost::thread_specific_ptr required that **Boost.Threads** be dynamically linked in order for its cleanup functionality to work on Win32 platforms. Because this limitation never applied to non−Win32 platforms, because significant progress has been made in removing the limitation on Win32 platforms (many thanks to Aaron LaFramboise and Roland Scwarz!), and because the lack of static linking is one of the most common complaints of **Boost.Threads** users, this decision was reversed.

On non−Win32 platforms: To choose the dynamically linked version of **Boost.Threads** using Boost's auto−linking feature, #define BOOST_THREAD_USE_DLL; to choose the statically linked version, #define BOOST_THREAD_USE_LIB. If neither symbols is #defined, the default will be chosen. Currently the default is the statically linked version.

On Win32 platforms using VC++: Use the same #defines as for non−Win32 platforms (BOOST_THREAD_USE_DLL and BOOST_THREAD_USE_LIB). If neither is #defined, the default will be chosen. Currently the default is the statically linked version if the VC++ run−time library is set to "Multi−threaded" or "Multi−threaded Debug", and the dynamically linked version if the VC++ run−time library is set to "Multi−threaded DLL" or "Multi−threaded Debug DLL".

On Win32 platforms using compilers other than VC++: Use the same #defines as for non−Win32 platforms (BOOST_THREAD_USE_DLL and BOOST_THREAD_USE_LIB). If neither is #defined, the default will be chosen. Currently the default is the dynamically linked version because it has not yet been possible to implement automatic tss cleanup in the statically linked version for compilers other than VC++, although it is hoped that this will be possible in a future version of **Boost.Threads**. Note for advanced users: **Boost.Threads** provides several "hook" functions to allow users to experiment with the statically linked version on Win32 with compilers other than VC++. These functions are on_process_enter(), on_process_exit(), on_thread_enter(), and on_thread_exit(), and are defined in tls_hooks.cpp. See the comments in that file for more information.

10.1.3. Barrier functionality added

A new class, boost::barrier, was added.

10.1.4. Read/write mutex functionality added

New classes, boost::read_write_mutex, boost::try_read_write_mutex, and boost::timed_read_write_mutex were added.

Note

Since the read/write mutex and related classes are new, both interface and implementation are liable to change in future releases of **Boost.Threads**. The lock concepts and lock promotion in particular are still under discussion and very likely to change.

10.1.5. Thread−specific pointer functionality changed

The boost::thread_specific_ptr constructor now takes an optional pointer to a cleanup function that is called to release the thread−specific data that is being pointed to by boost::thread_specific_ptr objects.

Fixed: the number of available thread−specific storage "slots" is too small on some platforms.

Fixed: thread_specific_ptr::reset() doesn't check error returned by tss::set() (the tss::set() function now throws if it fails instead of returning an error code).

Fixed: calling boost::thread_specific_ptr::reset() or boost::thread_specific_ptr::release() causes double−delete: once when boost::thread_specific_ptr::reset() or boost::thread_specific_ptr::release() is called and once when boost::thread_specific_ptr::~thread_specific_ptr() is called.

10.1.6. Mutex implementation changed for Win32

On Win32, boost::mutex, boost::try_mutex, boost::recursive_mutex, and boost::recursive_try_mutex now use a Win32 critical section whenever possible; otherwise they use a Win32 mutex. As before, boost::timed_mutex and boost::recursive_timed_mutex use a Win32 mutex.

10.1.7. Windows CE support improved

Minor changes were made to make Boost.Threads work on Windows CE.

Glossary

Definitions are given in terms of the C++ Standard [ISO98]. References to the standard are in the form [1.2.3/4], which represents the section number, with the paragraph number following the "/".

Because the definitions are written in something akin to "standardese", they can be difficult to understand. The intent isn't to confuse, but rather to clarify the additional requirements **Boost.Threads** places on a C++ implementation as defined by the C++ Standard.

Thread

Thread is short for "thread of execution". A thread of execution is an execution environment [1.9/7] within the execution environment of a C++ program [1.9]. The main() function [3.6.1] of the program is the initial function of the initial thread. A program in a multithreading environment always has an initial thread even if the program explicitly creates no additional threads.

Unless otherwise specified, each thread shares all aspects of its execution environment with other threads in the program. Shared aspects of the execution environment include, but are not limited to, the following:

- Static storage duration (static, extern) objects [3.7.1].
- Dynamic storage duration (heap) objects [3.7.3]. Thus each memory allocation will return a unique addresses, regardless of the thread making the allocation request.
- Automatic storage duration (stack) objects [3.7.2] accessed via pointer or reference from another thread.
- Resources provided by the operating system. For example, files.
- The program itself. In other words, each thread is executing some function of the same program, not a totally different program.

Each thread has its own:

- Registers and current execution sequence (program counter) [1.9/5].
- Automatic storage duration (stack) objects [3.7.2].

Thread−safe

A program is thread−safe if it has no race conditions, does not deadlock, and has no priority failures.

Note that thread−safety does not necessarily imply efficiency, and than while some thread−safety violations can be determined statically at compile time, many thread−safety errors can only only be detected at runtime.

Thread State

During the lifetime of a thread, it shall be in one of the following states:

Table 9.26. Thread States

State	Description
Ready	Ready to run, but waiting for a processor.
Running	Currently executing on a processor. Zero or more threads may be running at any time, with a maximum equal to the number of processors.
Blocked	Waiting for some resource other than a processor which is not currently available, or for the completion of calls to library functions [1.9/6]. The term "waiting" is synonymous with "blocked"
Terminated	Finished execution but not yet detached or joined.

Thread state transitions shall occur only as specified:

Table 9.27. Thread States Transitions

From	To	Cause
[none]	Ready	Thread is created by a call to a library function. In the case of the initial thread, creation is implicit and occurs during the startup of the main() function [3.6.1].
Ready	Running	Processor becomes available.
Running	Ready	Thread preempted.
Running	Blocked	Thread calls a library function which waits for a resource or for the completion of I/O.
Running	Terminated	Thread returns from its initial function, calls a thread termination library function, or is canceled by some other thread calling a thread termination library function.
Blocked	Ready	The resource being waited for becomes available, or the blocking library function completes.

Terminated	[none]	Thread is detached or joined by some other thread calling the appropriate library function, or by program termination [3.6.3].

[Note: if a suspend() function is added to the threading library, additional transitions to the blocked state will have to be added to the above table.]

Race Condition

A race condition is what occurs when multiple threads read from and write to the same memory without proper synchronization, resulting in an incorrect value being read or written. The result of a race condition may be a bit pattern which isn't even a valid value for the data type. A race condition results in undefined behavior [1.3.12].

Race conditions can be prevented by serializing memory access using the tools provided by **Boost.Threads**.

Deadlock

Deadlock is an execution state where for some set of threads, each thread in the set is blocked waiting for some action by one of the other threads in the set. Since each is waiting on the others, none will ever become ready again.

Starvation

The condition in which a thread is not making sufficient progress in its work during a given time interval.

Priority Failure

A priority failure (such as priority inversion or infinite overtaking) occurs when threads are executed in such a sequence that required work is not performed in time to be useful.

Undefined Behavior

The result of certain operations in **Boost.Threads** is undefined; this means that those operations can invoke almost any behavior when they are executed.

An operation whose behavior is undefined can work "correctly" in some implementations (i.e., do what the programmer thought it would do), while in other implementations it may exhibit almost any "incorrect" behavior−−such as returning an invalid value, throwing an exception, generating an access violation, or terminating the process.

Executing a statement whose behavior is undefined is a programming error.

Memory Visibility

An address [1.7] shall always point to the same memory byte, regardless of the thread or processor dereferencing the address.

An object [1.8, 1.9] is accessible from multiple threads if it is of static storage duration (static, extern) [3.7.1], or if a pointer or reference to it is explicitly or implicitly dereferenced in multiple threads.

For an object accessible from multiple threads, the value of the object accessed from one thread may be indeterminate or different from the value accessed from another thread, except under the conditions specified in the following table. For the same row of the table, the value of an object accessible at the indicated sequence point in thread A will be determinate and the same if accessed at or after the indicated sequence point in thread B, provided the object is not otherwise modified. In the table, the "sequence point at a call" is the sequence point after the evaluation of all function arguments [1.9/17], while the "sequence point after a call" is the sequence point after the copying of the returned value... [1.9/17].

Table 9.28. Memory Visibility

Thread A	Thread B
The sequence point at a call to a library thread−creation function.	The first sequence point of the initial function in the new thread created by the Thread A call.
The sequence point at a call to a library function which locks a mutex, directly or by waiting for a condition variable.	The sequence point after a call to a library function which unlocks the same mutex.

The last sequence point before thread termination.	The sequence point after a call to a library function which joins the terminated thread.
The sequence point at a call to a library function which signals or broadcasts a condition variable.	The sequence point after the call to the library function which was waiting on that same condition variable or signal.

The architecture of the execution environment and the observable behavior of the abstract machine [1.9] shall be the same on all processors.

The latitude granted by the C++ standard for an implementation to alter the definition of observable behavior of the abstract machine to include additional library I/O functions [1.9/6] is extended to include threading library functions.

When an exception is thrown and there is no matching exception handler in the same thread, behavior is undefined. The preferred behavior is the same as when there is no matching exception handler in a program [15.3/9]. That is, terminate() is called, and it is implementation−defined whether or not the stack is unwound.

11. Acknowledgements

William E. Kempf was the architect, designer, and implementor of **Boost.Threads**.

Mac OS Carbon implementation written by Mac Murrett.

Dave Moore provided initial submissions and further comments on the `barrier` , `thread_pool` , `read_write_mutex` , `read_write_try_mutex` and `read_write_timed_mutex` classes.

Important contributions were also made by Jeremy Siek (lots of input on the design and on the implementation), Alexander Terekhov (lots of input on the Win32 implementation, especially in regards to boost::condition, as well as a lot of explanation of POSIX behavior), Greg Colvin (lots of input on the design), Paul Mclachlan, Thomas Matelich and Iain Hanson (for help in trying to get the build to work on other platforms), and Kevin S. Van Horn (for several updates/corrections to the documentation).

Mike Glassford finished changes to **Boost.Threads** that were begun by William Kempf and moved them into the main CVS branch. He also addressed a number of issues that were brought up on the Boost developer's mailing list and provided some additions and changes to the read_write_mutex and related classes.

The documentation was written by William E. Kempf. Beman Dawes provided additional documentation material and editing. Mike Glassford finished William Kempf's conversion of the documentation to BoostBook format and added a number of new sections.

Discussions on the boost.org mailing list were essential in the development of **Boost.Threads** . As of August 1, 2001, participants included Alan Griffiths, Albrecht Fritzsche, Aleksey Gurtovoy, Alexander Terekhov, Andrew Green, Andy Sawyer, Asger Alstrup Nielsen, Beman Dawes, Bill Klein, Bill Rutiser, Bill Wade, Branko èibej, Brent Verner, Craig Henderson, Csaba Szepesvari, Dale Peakall, Damian Dixon, Dan Nuffer, Darryl Green, Daryle Walker, David Abrahams, David Allan Finch, Dejan Jelovic, Dietmar Kuehl, Douglas Gregor, Duncan Harris, Ed Brey, Eric Swanson, Eugene Karpachov, Fabrice Truillot, Frank Gerlach, Gary Powell, Gernot Neppert, Geurt Vos, Ghazi Ramadan, Greg Colvin, Gregory Seidman, HYS, Iain Hanson, Ian Bruntlett, J Panzer, Jeff Garland, Jeff Paquette, Jens Maurer, Jeremy Siek, Jesse Jones, Joe Gottman, John (EBo) David, John Bandela, John Maddock, John Max Skaller, John Panzer, Jon Jagger , Karl Nelson, Kevlin Henney, KG Chandrasekhar, Levente Farkas, Lie−Quan Lee, Lois Goldthwaite, Luis Pedro Coelho, Marc Girod, Mark A. Borgerding, Mark Rodgers, Marshall Clow, Matthew Austern, Matthew Hurd, Michael D. Crawford, Michael H. Cox , Mike Haller, Miki Jovanovic, Nathan Myers, Paul Moore, Pavel Cisler, Peter Dimov, Petr Kocmid, Philip Nash, Rainer Deyke, Reid Sweatman, Ross Smith, Scott McCaskill, Shalom Reich, Steve Cleary, Steven Kirk, Thomas Holenstein, Thomas Matelich, Trevor Perrin, Valentin Bonnard, Vesa Karvonen, Wayne Miller, and William Kempf.

Apologies for anyone inadvertently missed.

Bibliography

[AndrewsSchnieder83] *ACM Computing Surveys*. Vol. 15. No. 1. March, 1983. Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming .

Good general background reading. Includes descriptions of Path Expressions, Message Passing, and Remote Procedure Call in addition to the basics

[Boost] The *Boost* world wide web site. http://www.boost.org.

Boost.Threads is one of many Boost libraries. The Boost web site includes a great deal of documentation and general information which applies to all Boost libraries. Current copies of the libraries including documentation and test programs may be downloaded from the web site.

[Hansen73] *ACM Computing Surveys*. Vol. 5. No. 4. December, 1983. Per Brinch. Concurrent Programming Concepts .

"This paper describes the evolution of language features for multiprogramming from event queues and semaphores to critical regions and monitors." Includes analysis of why events are considered error−prone. Also noteworthy because of an introductory quotation from Christopher Alexander; Brinch Hansen was years ahead of others in recognizing pattern concepts applied to software, too.

[Butenhof97] *Programming with POSIX Threads* . David R. Butenhof. Addison−WesleyCopyright © 1997. ISNB: 0−201−63392−2.

This is a very readable explanation of threads and how to use them. Many of the insights given apply to all multithreaded programming, not just POSIX Threads

[Hoare74] *Communications of the ACM*. Vol. 17. No. 10. October, 1974. Monitors: An Operating System Structuring Concept . C.A.R. Hoare. 549−557.

Hoare and Brinch Hansen's work on Monitors is the basis for reliable multithreading patterns. This is one of the most often referenced papers in all of computer science, and with good reason.

[ISO98] *Programming Language C++*. ISO/IEC. 14882:1998(E).

This is the official C++ Standards document. Available from the ANSI (American National Standards Institute) Electronic Standards Store.

[McDowellHelmbold89] *Communications of the ACM*. Vol. 21. No. 2. December, 1989. Charles E. McDowell. David P. Helmbold. *Debugging Concurrent Programs*.

Identifies many of the unique failure modes and debugging difficulties associated with concurrent programs.

[SchmidtPyarali] *Strategies for Implementing POSIX Condition Variables on Win32*. Douglas C. Schmidt and Irfan Pyarali. Department of Computer Science, Washington University, St. Louis, Missouri.

Rationale for understanding **Boost.Threads** condition variables. Note that Alexander Terekhov found some bugs in the implementation given in this article, so pthreads−win32 and **Boost.Threads** are even more complicated yet.

[SchmidtStalRohnertBuschmann] *Pattern−Oriented Architecture Volume 2*. Patterns for Concurrent and Networked Objects. POSA2. Douglas C. Schmidt, Michael, Hans Rohnert, and Frank Buschmann. WileyCopyright © 2000.

This is a very good explanation of how to apply several patterns useful for concurrent programming. Among the patterns documented is the Monitor Pattern mentioned frequently in the **Boost.Threads** documentation.

[Stroustrup] *The C++ Programming Language*. Special Edition. Addison−WesleyCopyright © 2000. ISBN: 0−201−70073−5.

The first book a C++ programmer should own. Note that the 3rd edition (and subsequent editions like the Special Edition) has been rewritten to cover the ISO standard language and library.

Chapter 10. Boost.Tribool

Douglas Gregor

<dgregor -at- cs.indiana.edu>

1. Introduction

The 3–state boolean library contains a single class, `boost::logic::tribool`, along with support functions and operator overloads that implement 3–state boolean logic.

2. Tutorial

2.1. Basic usage

The `tribool` class acts like the built–in `bool` type, but for 3–state boolean logic. The three states are `true`, `false`, and `indeterminate`, where the first two states are equivalent to those of the C++ `bool` type and the last state represents an unknown boolean value (that may be `true` or `false`, we don't know).

The `tribool` class supports conversion from `bool` values and literals along with its own `indeterminate` keyword:

```
tribool b(true);
b = false;
b = indeterminate;
tribool b2(b);
```

`tribool` supports conversions to `bool` for use in conditional statements. The conversion to `bool` will be `true` when the value of the `tribool` is always true, and `false` otherwise. Consequently, the following idiom may be used to determine which of the three states a `tribool` currently holds:

```
tribool b = some_operation();
if (b) {
  // b is true
}
else if (!b) {
  // b is false
}
else {
  // b is indeterminate
}
```

`tribool` supports the 3–state logic operators `!` (negation), `&&` (AND), and `||` (OR), with `bool` and `tribool` values. For instance:

```
tribool x = some_op();
tribool y = some_other_op();
if (x && y) {
  // both x and y are true
}
else if (!(x && y)) {
  // either x or y is false
}
else {
  // neither x nor y is false, but we don't know that both are true
  if (x || y) {
    // either x or y is true
  }
}
```

Similarly, tribool supports 3−state equality comparisons via the operators == and !=. These operators differ from "normal" equality operators in C++ because they return a tribool, because potentially we might not know the result of a comparison (try to compare true and indeterminate). For instance:

```
tribool x(true);
tribool y(indeterminate);

assert(x == x); // okay, x == x returns true
assert(x == true); // okay, can compare tribools and bools
```

The indeterminate keyword (representing the indeterminate tribool value) doubles as a function to check if the value of a tribool is indeterminate, e.g.,

```
tribool x = try_to_do_something_tricky();
if (indeterminate(x)) {
  // value of x is indeterminate
}
else {
  // report success or failure of x
}
```

2.2. Renaming the indeterminate state

Users may introduce additional keywords for the indeterminate value in addition to the implementation−supplied indeterminate using the BOOST_TRIBOOL_THIRD_STATE macro. For instance, the following macro instantiation (at the global scope) will introduce the keyword maybe as a synonym for indeterminate (also residing in the boost namespace):

```
BOOST_TRIBOOL_THIRD_STATE(maybe)
tribool x = maybe;
if (maybe(x)) { /* ... */ }
```

2.3. tribool input/output

tribool objects may be read from and written to streams by including the boost/logic/tribool_io.hpp header in a manner very similar to bool values. When the boolalpha flag is not set on the input/output stream, the integral values 0, 1, and 2 correspond to tribool values false, true, and indeterminate, respectively. When boolalpha is set on the stream, arbitrary strings can be used to represent the three values, the default being "false", "true", and "indeterminate". For instance:

```
tribool x;
cin >> x; // Type "0", "1", or "2" to get false, true, or indeterminate
cout << boolalpha << x; // Produces "false", "true", or "indeterminate"
```

tribool input and output is sensitive to the stream's current locale. The strings associated with false and true values are contained in the standard std::numpunct facet, and the string naming the indeterminate type is contained in the indeterminate_name facet. To replace the name of the indeterminate state, you need to imbue your stream with a local containing a indeterminate_name facet, e.g.:

```
BOOST_TRIBOOL_THIRD_STATE(maybe)
locale global;
locale test_locale(global, new indeterminate_name<char>("maybe"));
cout.imbue(test_locale);
tribool x(maybe);
cout << boolalpha << x << endl; // Prints "maybe"
```

If you C++ standard library implementation does not support locales, `tribool` input/output will still work, but you will be unable to customize the strings printed/parsed when `boolalpha` is set.

3. Reference

3.1. Header <boost/logic/tribool.hpp>

```
BOOST_TRIBOOL_THIRD_STATE(Name)

namespace boost {
  namespace logic {
    class tribool;
    bool indeterminate(tribool, unspecified = unspecified);
    tribool operator!(tribool);
    tribool operator&&(tribool, tribool);
    tribool operator&&(tribool, bool);
    tribool operator&&(bool, tribool);
    tribool operator&&(indeterminate_keyword_t, tribool);
    tribool operator&&(tribool, indeterminate_keyword_t);
    tribool operator||(tribool, tribool);
    tribool operator||(tribool, bool);
    tribool operator||(bool, tribool);
    tribool operator||(indeterminate_keyword_t, tribool);
    tribool operator||(tribool, indeterminate_keyword_t);
    tribool operator==(tribool, tribool);
    tribool operator==(tribool, bool);
    tribool operator==(bool, tribool);
    tribool operator==(indeterminate_keyword_t, tribool);
    tribool operator==(tribool, indeterminate_keyword_t);
    tribool operator!=(tribool, tribool);
    tribool operator!=(tribool, bool);
    tribool operator!=(bool, tribool);
    tribool operator!=(indeterminate_keyword_t, tribool);
    tribool operator!=(tribool, indeterminate_keyword_t);
  }
}
```

Class tribool

boost::logic::tribool — A 3−state boolean type.

Synopsis

```
class tribool {
public:
  // construct/copy/destruct
  tribool();
  tribool(bool);
  tribool(indeterminate_keyword_t);

  // public member functions
  operator safe_bool() const;

  enum boost::logic::tribool::@0 value;
};
```

Description

3−state boolean values are either true, false, or indeterminate.

`tribool` construct/copy/destruct

1. `tribool();`

   Construct a new 3−state boolean value with the value 'false'.

   Throws
   : Will not throw.

2. `tribool(bool value);`

   Construct a new 3−state boolean value with the given boolean value, which may be `true` or `false` .

   Throws
   : Will not throw.

3. `tribool(indeterminate_keyword_t );`

   Construct a new 3−state boolean value with an indeterminate value.

   Throws
   : Will not throw.

`tribool` public member functions

1. `operator safe_bool() const;`

   Use a 3−state boolean in a boolean context. Will evaluate true in a boolean context only when the 3−state boolean is definitely true.

   Returns
   : true if the 3−state boolean is true, false otherwise

   Throws
   : Will not throw.

Function indeterminate

boost::logic::indeterminate   Keyword and test function for the indeterminate tribool value.

Synopsis

```
bool indeterminate(tribool x, unspecified dummy = unspecified);
```

Description

The `indeterminate` function has a dual role. It's first role is as a unary function that tells whether the tribool value is in the "indeterminate" state. It's second role is as a keyword representing the indeterminate (just like "true" and "false" represent the true and false states). If you do not like the name "indeterminate", and would prefer to use a different name, see the macro `BOOST_TRIBOOL_THIRD_STATE` .

Returns
: `x.value == tribool::indeterminate_value`

Throws
: Will not throw.

Function operator!

boost::logic::operator! — Computes the logical negation of a tribool.

Synopsis

```
tribool operator!(tribool x);
```

Description

Returns

the logical negation of the tribool, according to the table:

!	
false	true
true	false
indeterminate	indeterminate

Throws

Will not throw.

Function operator&&

boost::logic::operator&& — Computes the logical conjuction of two tribools.

Synopsis

```
tribool operator&&(tribool x, tribool y);
tribool operator&&(tribool x, bool y);
tribool operator&&(bool x, tribool y);
tribool operator&&(indeterminate_keyword_t , tribool x);
tribool operator&&(tribool x, indeterminate_keyword_t );
```

Description

Returns

the result of logically ANDing the two tribool values, according to the following table:

&&	**false**	**true**	**indeterminate**
false	false	false	false
true	false	true	indeterminate
indeterminate	false	indeterminate	indeterminate

Throws

Will not throw.

Function operator||

boost::logic::operator|| — Computes the logical disjunction of two tribools.

Synopsis

```
tribool operator||(tribool x, tribool y);
tribool operator||(tribool x, bool y);
tribool operator||(bool x, tribool y);
tribool operator||(indeterminate_keyword_t , tribool x);
tribool operator||(tribool x, indeterminate_keyword_t );
```

Description

Returns
: the result of logically ORing the two tribool values, according to the following table:

||	**false**	**true**	**indeterminate**
false	false	true	indeterminate
true	true	true	true
indeterminate	indeterminate	true	indeterminate

Throws
: Will not throw.

Function operator==

boost::logic::operator== Compare tribools for equality.

Synopsis

```
tribool operator==(tribool x, tribool y);
tribool operator==(tribool x, bool y);
tribool operator==(bool x, tribool y);
tribool operator==(indeterminate_keyword_t , tribool x);
tribool operator==(tribool x, indeterminate_keyword_t );
```

Description

Returns
: the result of comparing two tribool values, according to the following table:

==	**false**	**true**	**indeterminate**
false	true	false	indeterminate
true	false	true	indeterminate
indeterminate	indeterminate	indeterminate	indeterminate

Throws
: Will not throw.

Function operator!=

boost::logic::operator!= Compare tribools for inequality.

Synopsis

```
tribool operator!=(tribool x, tribool y);
tribool operator!=(tribool x, bool y);
tribool operator!=(bool x, tribool y);
tribool operator!=(indeterminate_keyword_t , tribool x);
tribool operator!=(tribool x, indeterminate_keyword_t );
```

Description

Returns
: the result of comparing two tribool values for inequality, according to the following table:

!=	false	true	indeterminate
false	false	true	indeterminate
true	true	false	indeterminate
indeterminate	indeterminate	indeterminate	indeterminate

Throws
: Will not throw.

Macro BOOST_TRIBOOL_THIRD_STATE

BOOST_TRIBOOL_THIRD_STATE — Declare a new name for the third state of a tribool.

Synopsis

```
BOOST_TRIBOOL_THIRD_STATE(Name)
```

Description

Use this macro to declare a new name for the third state of a tribool. This state can have any number of new names (in addition to `indeterminate`), all of which will be equivalent. The new name will be placed in the namespace in which the macro is expanded.

Example: BOOST_TRIBOOL_THIRD_STATE(true_or_false)

tribool x(true_or_false); // potentially set x if (true_or_false(x)) { // don't know what x is }

3.2. Header <boost/logic/tribool_fwd.hpp>

```
namespace boost {
  namespace logic {
  }
}
```

3.3. Header <boost/logic/tribool_io.hpp>

```
namespace boost {
  namespace logic {
    template<typename CharT> class indeterminate_name;
    template<typename T>
      std::basic_string< T > get_default_indeterminate_name();

    // Returns the character string "indeterminate".
    template<> 
      std::basic_string< char > get_default_indeterminate_name<char >();

    // Returns the wide character string L"indeterminate". 
    template<> 
      std::basic_string< wchar_t > get_default_indeterminate_name<wchar_t >();
    template<typename CharT, typename Traits> 
      std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, tribool);
    template<typename CharT, typename Traits> 
      std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, tribool &);
  }
}
```

Class template indeterminate_name

boost::logic::indeterminate_name — A locale facet specifying the name of the indeterminate value of a tribool.

Synopsis

```
template<typename CharT> 
class indeterminate_name {
public:
  // types
  typedef CharT                         char_type;  
  typedef std::basic_string< CharT > string_type;

  // construct/copy/destruct
  indeterminate_name();
  indeterminate_name(const string_type &);

  // public member functions
  string_type name() const;

  static std::locale::id id;
};
```

Description

The facet is used to perform I/O on tribool values when `std::boolalpha` has been specified. This class template is only available if the C++ standard library implementation supports locales.

`indeterminate_name` construct/copy/destruct

1. `indeterminate_name();`
2. `indeterminate_name(const string_type & name);`

`indeterminate_name` public member functions

1. `string_type name() const;`

Function template get_default_indeterminate_name

boost::logic::get_default_indeterminate_name — Returns a string containing the default name for the indeterminate value of a tribool with the given character type T.

Synopsis

```
template<typename T> std::basic_string< T > get_default_indeterminate_name();
```

Description

This routine is used by the input and output streaming operators for tribool when there is no locale support or the stream's locale does not contain the indeterminate_name facet.

Function template operator<<

boost::logic::operator<< — Writes the value of a tribool to a stream.

Synopsis

```
template<typename CharT, typename Traits>
  std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & out, tribool x);
```

Description

When the value of `x` is either `true` or `false` , this routine is semantically equivalent to:

```
 out << static_cast<bool>(x);
```

When `x` has an indeterminate value, it outputs either the integer value 2 (if `(out.flags() & std::ios_base::boolalpha) == 0`) or the name of the indeterminate value. The name of the indeterminate value comes from the indeterminate_name facet (if it is defined in the output stream's locale), or from the get_default_indeterminate_name function (if it is not defined in the locale or if the C++ standard library implementation does not support locales).

Returns
: `out`

Function template operator>>

boost::logic::operator>> — Reads a tribool value from a stream.

Synopsis

```
template<typename CharT, typename Traits>
  std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & in, tribool & x);
```

Description

When `(out.flags() & std::ios_base::boolalpha) == 0` , this function reads a `long` value from the input stream `in` and converts that value to a tribool. If that value is 0, `x` becomes `false` ; if it is 1, `x` becomes `true` ; if it is 2, `becomesindetermine` ; otherwise, the operation fails (and the fail bit is set on the input stream `in`).

When `(out.flags() & std::ios_base::boolalpha) != 0` , this function first determines the names of the false, true, and indeterminate values. The false and true names are extracted from the `std::numpunct` facet of the input stream's locale (if the C++ standard library implementation supports locales), or from the `default_false_name` and `default_true_name` functions (if there is no locale support). The indeterminate name is extracted from the appropriate `indeterminate_name` facet (if it is available in the input stream's locale), or from the `get_default_indeterminate_name` function (if the C++ standard library implementation does not support locales, or the `indeterminate_name` facet is not specified for this locale object). The input is then matched to each of these names, and the tribool `x` is assigned the value corresponding to the longest name that matched. If no name is matched or all names are empty, the operation fails (and the fail bit is set on the input stream `in`).

Returns
    `in`

4. Testsuite

4.1. Acceptance tests

Test	Type	Description	If failing...
tribool_test.cpp	run	Test all features of the `boost::logic::tribool` class.	
tribool_rename_test.cpp	run	Test the use of the `BOOST_TRIBOOL_THIRD_STATE` macro.	
tribool_io_test.cpp	run	Test tribool input/output.	

Chapter 11. Boost.Variant

Eric Friedman

Itay Maman

1. Introduction

1.1. Abstract

The `variant` class template is a safe, generic, stack−based discriminated union container, offering a simple solution for manipulating an object from a heterogeneous set of types in a uniform manner. Whereas standard containers such as `std::vector` may be thought of as "**multi−value, single type**," `variant` is "**multi−type, single value**."

Notable features of `boost::variant` include:

- Full value semantics, including adherence to standard overload resolution rules for conversion operations.
- Compile−time type−safe value visitation via `boost::apply_visitor`.
- Run−time checked explicit value retrieval via `boost::get`.
- Support for recursive variant types via both `boost::make_recursive_variant` and `boost::recursive_wrapper`.
- Efficient implementation −− stack−based when possible (see Section 4.1, "Never−Empty" Guarantee for more details).

1.2. Motivation

1.2.1. Problem

Many times, during the development of a C++ program, the programmer finds himself in need of manipulating several distinct types in a uniform manner. Indeed, C++ features direct language support for such types through its `union` keyword:

```
union { int i; double d; } u;
u.d = 3.14;
u.i = 3; // overwrites u.d (OK: u.d is a POD type)
```

C++'s `union` construct, however, is nearly useless in an object−oriented environment. The construct entered the language primarily as a means for preserving compatibility with C, which supports only POD (Plain Old Data) types, and so does not accept types exhibiting non−trivial construction or destruction:

```
union {
  int i;
  std::string s; // illegal: std::string is not a POD type!
} u;
```

Clearly another approach is required. Typical solutions feature the dynamic−allocation of objects, which are subsequently manipulated through a common base type (often a virtual base class [Hen01] or, more dangerously, a `void*`). Objects of concrete type may be then retrieved by way of a polymorphic downcast construct (e.g., `dynamic_cast`, boost::any_cast, etc.).

However, solutions of this sort are highly error–prone, due to the following:

- *Downcast errors cannot be detected at compile–time.* Thus, incorrect usage of downcast constructs will lead to bugs detectable only at run–time.
- *Addition of new concrete types may be ignored.* If a new concrete type is added to the hierarchy, existing downcast code will continue to work as–is, wholly ignoring the new type. Consequently, the programmer must manually locate and modify code at numerous locations, which often results in run–time errors that are difficult to find.

Furthermore, even when properly implemented, these solutions tend to incur a relatively significant abstraction penalty due to the use of the heap, virtual function calls, and polymorphic downcasts.

1.2.2. Solution: A Motivating Example

The `boost::variant` class template addresses these issues in a safe, straightforward, and efficient manner. The following example demonstrates how the class can be used:

```
#include "boost/variant.hpp"
#include <iostream>

class my_visitor : public boost::static_visitor<int>
{
public:
    int operator()(int i) const
    {
        return i;
    }

    int operator()(const std::string & str) const
    {
        return str.length();
    }
};

int main()
{
    boost::variant< int, std::string > u("hello world");
    std::cout << u; // output: hello world

    int result = boost::apply_visitor( my_visitor(), u );
    std::cout << result; // output: 11 (i.e., length of "hello world")
}
```

2. Tutorial

2.1. Basic Usage

A discriminated union container on some set of types is defined by instantiating the `boost::variant` class template with the desired types. These types are called **bounded types** and are subject to the requirements of the *BoundedType* concept. Any number of bounded types may be specified, up to some implementation–defined limit (see `BOOST_VARIANT_LIMIT_TYPES`).

For example, the following declares a discriminated union container on `int` and `std::string`:

```
boost::variant< int, std::string > v;
```

By default, a `variant` default–constructs its first bounded type, so `v` initially contains `int(0)`. If this is not desired, or if the first bounded type is not default–constructible, a `variant` can be constructed directly from any value convertible to one of its bounded types. Similarly, a `variant` can be assigned any value convertible to one of its bounded types, as demonstrated in the following:

```
v = "hello";
```

Now `v` contains a `std::string` equal to `"hello"`. We can demonstrate this by **streaming** `v` to standard output:

```
std::cout << v << std::endl;
```

Usually though, we would like to do more with the content of a `variant` than streaming. Thus, we need some way to access the contained value. There are two ways to accomplish this: `apply_visitor`, which is safest and very powerful, and `get<T>`, which is sometimes more convenient to use.

For instance, suppose we wanted to concatenate to the string contained in `v`. With **value retrieval** by `get`, this may be accomplished quite simply, as seen in the following:

```
std::string& str = boost::get<std::string>(v);
str += " world! ";
```

As desired, the `std::string` contained by `v` now is equal to `"hello world! "`. Again, we can demonstrate this by streaming `v` to standard output:

```
std::cout << v << std::endl;
```

While use of `get` is perfectly acceptable in this trivial example, `get` generally suffers from several significant shortcomings. For instance, if we were to write a function accepting a `variant<int, std::string>`, we would not know whether the passed `variant` contained an `int` or a `std::string`. If we insisted upon continued use of `get`, we would need to query the `variant` for its contained type. The following function, which "doubles" the content of the given `variant`, demonstrates this approach:

```
void times_two( boost::variant< int, std::string > & operand )
{
    if ( int* pi = boost::get<int>( &v ) )
        *pi *= 2;
    else if ( std::string* pstr = boost::get<std::string>( &v ) )
        *pstr += *pstr;
}
```

However, such code is quite brittle, and without careful attention will likely lead to the introduction of subtle logical errors detectable only at runtime. For instance, consider if we wished to extend `times_two` to operate on a `variant` with additional bounded types. Specifically, let's add `std::complex<double>` to the set. Clearly, we would need to at least change the function declaration:

```
void times_two( boost::variant< int, std::string, std::complex<double> > & operand )
{
    // as above...?
}
```

Of course, additional changes are required, for currently if the passed `variant` in fact contained a `std::complex` value, `times_two` would silently return −− without any of the desired side−effects and without any error. In this case, the fix is obvious. But in more complicated programs, it could take considerable time to identify and locate the error in the first place.

Thus, real−world use of `variant` typically demands an access mechanism more robust than `get`. For this reason, `variant` supports compile−time checked **visitation** via `apply_visitor`. Visitation requires that the programmer explicitly handle (or ignore) each bounded type. Failure to do so results in a compile−time error.

Visitation of a `variant` requires a visitor object. The following demonstrates one such implementation of a visitor implementating behavior identical to `times_two`:

```
class times_two_visitor
    : public boost::static_visitor<>
```

```
{
public:

    void operator()(int & i) const
    {
        i *= 2;
    }

    void operator()(std::string & str) const
    {
        str += str;
    }

};
```

With the implementation of the above visitor, we can then apply it to `v`, as seen in the following:

```
boost::apply_visitor( times_two_visitor(), v );
```

As expected, the content of `v` is now a `std::string` equal to `"hello world! hello world! "`. (We'll skip the verification this time.)

In addition to enhanced robustness, visitation provides another important advantage over `get`: the ability to write generic visitors. For instance, the following visitor will "double" the content of *any* `variant` (provided its bounded types each support operator+=):

```
class times_two_generic
    : public boost::static_visitor<>
{
public:

    template <typename T>
    void operator()( T & operand ) const
    {
        operand += operand;
    }

};
```

Again, `apply_visitor` sets the wheels in motion:

```
boost::apply_visitor( times_two_generic(), v );
```

While the initial setup costs of visitation may exceed that required for `get`, the benefits quickly become significant. Before concluding this section, we should explore one last benefit of visitation with `apply_visitor`: **delayed visitation**. Namely, a special form of `apply_visitor` is available that does not immediately apply the given visitor to any `variant` but rather returns a function object that operates on any `variant` given to it. This behavior is particularly useful when operating on sequences of `variant` type, as the following demonstrates:

```
std::vector< boost::variant<int, std::string> > vec;
vec.push_back( 21 );
vec.push_back( "hello " );

times_two_generic visitor;
std::for_each(
      vec.begin(), vec.end()
   , boost::apply_visitor(visitor)
   );
```

2.2. Advanced Topics

This section discusses several features of the library often required for advanced uses of `variant`. Unlike in the above section, each feature presented below is largely independent of the others. Accordingly, this section is not necessarily intended to be read linearly or in its entirety.

2.2.1. Preprocessor macros

While the `variant` class template's variadic parameter list greatly simplifies use for specific instantiations of the template, it significantly complicates use for generic instantiations. For instance, while it is immediately clear how one might write a function accepting a specific `variant` instantiation, say `variant<int, std::string>`, it is less clear how one might write a function accepting any given `variant`.

Due to the lack of support for true variadic template parameter lists in the C++98 standard, the preprocessor is needed. While the Preprocessor library provides a general and powerful solution, the need to repeat `BOOST_VARIANT_LIMIT_TYPES` unnecessarily clutters otherwise simple code. Therefore, for common use–cases, this library provides its own macro **`BOOST_VARIANT_ENUM_PARAMS`**.

This macro simplifies for the user the process of declaring `variant` types in function templates or explicit partial specializations of class templates, as shown in the following:

```
// general cases
template <typename T> void some_func(const T &);
template <typename T> class some_class;

// function template overload
template <BOOST_VARIANT_ENUM_PARAMS(typename T)>
void some_func(const boost::variant<BOOST_VARIANT_ENUM_PARAMS(T)> &);

// explicit partial specialization
template <BOOST_VARIANT_ENUM_PARAMS(typename T)>
class some_class< boost::variant<BOOST_VARIANT_ENUM_PARAMS(T)> >;
```

2.2.2. Using a type sequence to specify bounded types

While convenient for typical uses, the `variant` class template's variadic template parameter list is limiting in two significant dimensions. First, due to the lack of support for true variadic template parameter lists in C++, the number of parameters must be limited to some implementation–defined maximum (namely, `BOOST_VARIANT_LIMIT_TYPES`). Second, the nature of parameter lists in general makes compile–time manipulation of the lists excessively difficult.

To solve these problems, `make_variant_over<` *`Sequence`* `>` exposes a `variant` whose bounded types are the elements of `Sequence` (where `Sequence` is any type fulfilling the requirements of MPL's *Sequence* concept). For instance,

```
typedef mpl::vector< std::string > types_initial;
typedef mpl::push_front< types_initial, int >::type types;

boost::make_variant_over< types >::type v1;
```

behaves equivalently to

```
boost::variant< int, std::string > v2;
```

Portability: Unfortunately, due to standard conformance issues in several compilers, `make_variant_over` is not universally available. On these compilers the library indicates its lack of support for the syntax via the definition of the preprocessor symbol `BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT`.

2.2.3. Recursive `variant` types

Recursive types facilitate the construction of complex semantics from simple syntax. For instance, nearly every programmer is familiar with the canonical definition of a linked list implementation, whose simple definition allows sequences of unlimited length:

```
template <typename T>
struct list_node
{
    T data;
    list_node * next;
};
```

The nature of `variant` as a generic class template unfortunately precludes the straightforward construction of recursive `variant` types. Consider the following attempt to construct a structure for simple mathematical expressions:

```
struct add;
struct sub;
template <typename OpTag> struct binary_op;

typedef boost::variant<
      int
    , binary_op<add>
    , binary_op<sub>
    > expression;

template <typename OpTag>
struct binary_op
{
    expression left;  // variant instantiated here...
    expression right;

    binary_op( const expression & lhs, const expression & rhs )
        : left(lhs), right(rhs)
    {
    }

}; // ...but binary_op not complete until here!
```

While well–intentioned, the above approach will not compile because `binary_op` is still incomplete when the `variant` type `expression` is instantiated. Further, the approach suffers from a more significant logical flaw: even if C++ syntax were different such that the above example could be made to "work," `expression` would need to be of infinite size, which is clearly impossible.

To overcome these difficulties, `variant` includes special support for the `boost::recursive_wrapper` class template, which breaks the circular dependency at the heart of these problems. Further, `boost::make_recursive_variant` provides a more convenient syntax for declaring recursive `variant` types. Tutorials for use of these facilities is described in Section 2.2.3.1, Recursive types with `recursive_wrapper` and Section 2.2.3.2, Recursive types with `make_recursive_variant` .

2.2.3.1. Recursive types with `recursive_wrapper`

The following example demonstrates how `recursive_wrapper` could be used to solve the problem presented in Section 2.2.3, Recursive `variant` types :

```
typedef boost::variant<
      int
    , boost::recursive_wrapper< binary_op<add> >
    , boost::recursive_wrapper< binary_op<sub> >
    > expression;
```

Because `variant` provides special support for `recursive_wrapper`, clients may treat the resultant `variant` as though the wrapper were not present. This is seen in the implementation of the following visitor, which calculates the value of an `expression` without any reference to `recursive_wrapper`:

```
class calculator : public boost::static_visitor<int>
{
public:
    int operator()(int value) const
    {
        return value;
    }

    int operator()(const binary_op<add> & binary) const
    {
        return boost::apply_visitor( calculator(), binary.left )
             + boost::apply_visitor( calculator(), binary.right );
    }

    int operator()(const binary_op<sub> & binary) const
    {
        return boost::apply_visitor( calculator(), binary.left )
             - boost::apply_visitor( calculator(), binary.right );
    }

};
```

Finally, we can demonstrate `expression` in action:

```
void f()
{
    // result = ((7-3)+8) = 12
    expression result(
        binary_op<add>(
            binary_op<sub>(7,3)
          , 8
          )
      );

    assert( boost::apply_visitor(calculator(),result) == 12 );
}
```

2.2.3.2. Recursive types with `make_recursive_variant`

For some applications of recursive `variant` types, a user may be able to sacrifice the full flexibility of using `recursive_wrapper` with `variant` for the following convenient syntax:

```
typedef boost::make_recursive_variant<
      int
    , std::vector< boost::recursive_variant_ >
    >::type int_tree_t;
```

Use of the resultant `variant` type is as expected:

```
std::vector< int_tree_t > subresult;
subresult.push_back(3);
subresult.push_back(5);

std::vector< int_tree_t > result;
result.push_back(1);
result.push_back(subresult);
result.push_back(7);

int_tree_t var(result);
```

To be clear, one might represent the resultant content of `var` as `( 1 ( 3 5 ) 7 )`.

Finally, note that a type sequence can be used to specify the bounded types of a recursive `variant` via the use of `boost::make_recursive_variant_over`, whose semantics are the same as `make_variant_over` (which is described in Section 2.2.2, "Using a type sequence to specify bounded types").

Portability: Unfortunately, due to standard conformance issues in several compilers, `make_recursive_variant` is not universally supported. On these compilers the library indicates its lack of support via the definition of the preprocessor symbol `BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT`. Thus, unless working with highly−conformant compilers, maximum portability will be achieved by instead using `recursive_wrapper`, as described in Section 2.2.3.1, Recursive types with `recursive_wrapper` .

2.2.4. Binary visitation

As the tutorial above demonstrates, visitation is a powerful mechanism for manipulating `variant` content. Binary visitation further extends the power and flexibility of visitation by allowing simultaneous visitation of the content of two different `variant` objects.

Notably this feature requires that binary visitors are incompatible with the visitor objects discussed in the tutorial above, as they must operate on two arguments. The following demonstrates the implementation of a binary visitor:

```
class are_strict_equals
    : public boost::static_visitor<bool>
{
public:

    template <typename T, typename U>
    bool operator()( const T &, const U & )
    {
        return false; // cannot compare different types
    }

    template <typename T>
    bool operator()( const T & lhs, const T & rhs )
    {
        return lhs == rhs;
    }

};
```

As expected, the visitor is applied to two `variant` arguments by means of `apply_visitor`:

```
boost::variant< int, std::string > v1( "hello" );

boost::variant< double, std::string > v2( "hello" );
assert( boost::apply_visitor(are_strict_equals(), v1, v2) );

boost::variant< int, const char * > v3( "hello" );
assert( !boost::apply_visitor(are_strict_equals(), v1, v3) );
```

Finally, we must note that the function object returned from the "delayed" form of `apply_visitor` also supports binary visitation, as the following demonstrates:

```
typedef boost::variant<double, std::string> my_variant;

std::vector< my_variant > seq1;
seq1.push_back("pi is close to ");
seq1.push_back(3.14);

std::list< my_variant > seq2;
seq2.push_back("pi is close to ");
seq2.push_back(3.14);

are_strict_equals visitor;
assert( std::equal(
      v1.begin(), v1.end(), v2.begin()
    , boost::apply_visitor( visitor )
    ) );
```

3. Reference

3.1. Concepts

3.1.1. *BoundedType*

The requirements on a **bounded type** are as follows:

- CopyConstructible [20.1.3].
- Destructor upholds the no−throw exception−safety guarantee.
- Complete at the point of `variant` template instantiation. (See `boost::recursive_wrapper<T>` for a type wrapper that accepts incomplete types to enable recursive `variant` types.)

Every type specified as a template argument to `variant` must at minimum fulfill the above requirements. In addition, certain features of `variant` are available only if its bounded types meet the requirements of these following additional concepts:

- Assignable: `variant` is itself *Assignable* if and only if every one of its bounded types meets the requirements of the concept. (Note that top−level `const`−qualified types and reference types do *not* meet these requirements.)
- DefaultConstructible [20.1.4]: `variant` is itself DefaultConstructible if and only if its first bounded type (i.e., `T1`) meets the requirements of the concept.
- EqualityComparable: `variant` is itself EqualityComparable if and only if every one of its bounded types meets the requirements of the concept.
- LessThanComparable: `variant` is itself LessThanComparable if and only if every one of its bounded types meets the requirements of the concept.
- *OutputStreamable*: `variant` is itself *OutputStreamable* if and only if every one of its bounded types meets the requirements of the concept.

3.1.2. *StaticVisitor*

The requirements on a **static visitor** of a type `T` are as follows:

- Must allow invocation as a function by overloading `operator()`, unambiguously accepting any value of type `T`.
- Must expose inner type `result_type`. (See `boost::visitor_ptr` for a solution to using functions as visitors.)
- If `result_type` is not `void`, then each operation of the function object must return a value implicitly convertible to `result_type`.

3.1.2.1. Examples

The following class satisfies the requirements of a static visitor of several types (i.e., explicitly: `int` and `std::string`; or, e.g., implicitly: `short` and `const char *`; etc.):

```
class my_visitor
    : public boost::static_visitor<int>
{
public:

    int operator()(int i)
    {
        return i * 2;
    }

    int operator()(const std::string& s)
    {
        return s.length();
    }

};
```

Another example is the following class, whose function−call operator is a member template, allowing it to operate on values of many types. Thus, the following class is a visitor of any type that supports streaming output (e.g., `int`, `double`, `std::string`, etc.):

```
class printer
    : public boost::static_visitor<>
{
    template <typename T>
    void operator()(const T& t)
    {
        std::cout << t << std::endl;
    }
};
```

3.1.3. *OutputStreamable*

The requirements on an **output streamable** type `T` are as follows:

- For any object `t` of type `T`, `std::cout << t` must be a valid expression.

3.2. Header <boost/variant.hpp>

This header exists simply as a convenience to the user, including all of the headers in the `boost/variant` directory.

3.3. Header <boost/variant/variant_fwd.hpp>

Provides forward declarations of the `boost::variant`, `boost::make_variant_over`, `boost::make_recursive_variant`, and `boost::make_recursive_variant_over` class templates and the `boost::recursive_variant_` tag type. Also defines several preprocessor symbols, as described below.

```
BOOST_VARIANT_LIMIT_TYPES
BOOST_VARIANT_ENUM_PARAMS(param)
BOOST_VARIANT_ENUM_SHIFTED_PARAMS(param)
BOOST_VARIANT_NO_REFERENCE_SUPPORT
BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT
BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT
```

Macro BOOST_VARIANT_LIMIT_TYPES

BOOST_VARIANT_LIMIT_TYPES

Expands to the length of the template parameter list for `variant`.

Synopsis

```
BOOST_VARIANT_LIMIT_TYPES
```

Description

Note: Conforming implementations of `variant` must allow at least ten template arguments. That is, BOOST_VARIANT_LIMIT_TYPES must be greater or equal to `10`.

Macro BOOST_VARIANT_ENUM_PARAMS

BOOST_VARIANT_ENUM_PARAMS

Enumerate parameters for use with `variant`.

Synopsis

```
BOOST_VARIANT_ENUM_PARAMS(param)
```

Description

Expands to a comma–separated sequence of length `BOOST_VARIANT_LIMIT_TYPES`, where each element in the sequence consists of the concatenation of *param* with its zero–based index into the sequence. That is, `param ## 0, param ## 1, ..., param ## BOOST_VARIANT_LIMIT_TYPES - 1`.

Rationale: This macro greatly simplifies for the user the process of declaring `variant` types in function templates or explicit partial specializations of class templates, as shown in the tutorial.

Macro BOOST_VARIANT_ENUM_SHIFTED_PARAMS

BOOST_VARIANT_ENUM_SHIFTED_PARAMS

Enumerate all but the first parameter for use with `variant`.

Synopsis

```
BOOST_VARIANT_ENUM_SHIFTED_PARAMS(param)
```

Description

Expands to a comma–separated sequence of length `BOOST_VARIANT_LIMIT_TYPES - 1`, where each element in the sequence consists of the concatenation of *param* with its one–based index into the sequence. That is, `param ## 1, ..., param ## BOOST_VARIANT_LIMIT_TYPES - 1`.

Note: This macro results in the same expansion as `BOOST_VARIANT_ENUM_PARAMS` –– but without the first term.

Macro BOOST_VARIANT_NO_REFERENCE_SUPPORT

BOOST_VARIANT_NO_REFERENCE_SUPPORT

Indicates `variant` does not support references as bounded types.

Synopsis

```
BOOST_VARIANT_NO_REFERENCE_SUPPORT
```

Description

Defined only if `variant` does not support references as bounded types.

Macro BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT

BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT

Indicates absence of support for specifying the bounded types of a `variant` by the elements of a type sequence.

Synopsis

```
BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT
```

Description

Defined only if `make_variant_over` and `make_recursive_variant_over` are not supported for some reason on the target compiler.

Macro BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT

BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT

Indicates `make_recursive_variant` operates in an implementation–defined manner.

Synopsis

```
BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT
```

Description

Defined only if `make_recursive_variant` does not operate as documented on the target compiler, but rather in an implementation–defined manner.

Implementation Note: If `BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT` is defined for the target compiler, the current implementation uses the MPL lambda mechanism to approximate the desired behavior. (In most cases, however, such compilers do not have full lambda support either.)

3.4. Header <boost/variant/variant.hpp>

```
namespace boost {
  template<typename T1, typename T2 = unspecified, ...,
           typename TN = unspecified>
    class variant;

  template<typename Sequence> class make_variant_over;
  template<typename T1, typename T2, ..., typename TN>
    void swap(variant<T1, T2, ..., TN> &, variant<T1, T2, ..., TN> &);
  template<typename ElemType, typename Traits, typename T1, typename T2, ...,
           typename TN>
    std::basic_ostream<ElemType,Traits> &
    operator<<(std::basic_ostream<ElemType,Traits> &,
               const variant<T1, T2, ..., TN> &);
}
```

Class template variant

boost::variant Safe, generic, stack–based discriminated union container.

Synopsis

```
template<typename T1, typename T2 = unspecified, ...,
         typename TN = unspecified>
class variant {
public:
  // types
  typedef unspecified types;

  // construct/copy/destruct
  variant();
  variant(const variant &);
  template<typename T> variant(T &);
  template<typename T> variant(const T &);
  template<typename U1, typename U2, ..., typename UN>
    variant(variant<U1, U2, ..., UN> &);
  template<typename U1, typename U2, ..., typename UN>
    variant(const variant<U1, U2, ..., UN> &);
  ~variant();

  // modifiers
  void swap(variant &);
  variant & operator=(const variant &);
  template<typename T> variant & operator=(const T &);

  // queries
  int which() const;
  bool empty() const;
  const std::type_info & type() const;

  // relational
  bool operator==(const variant &) const;
  template<typename U> void operator==(const U &) const;
  bool operator<(const variant &) const;
  template<typename U> void operator<(const U &) const;
};
```

Description

The `variant` class template (inspired by Andrei Alexandrescu's class of the same name [Ale01A]) is an efficient, recursive−capable, bounded discriminated union value type capable of containing any value type (either POD or non−POD). It supports construction from any type convertible to one of its bounded types or from a source `variant` whose bounded types are each convertible to one of the destination `variant`'s bounded types. As well, through `apply_visitor`, `variant` supports compile−time checked, type−safe visitation; and through `get`, `variant` supports run−time checked, type−safe value retrieval.

Notes:

- The bounded types of the `variant` are exposed via the nested typedef `types`, which is an MPL−compatible Sequence containing the set of types that must be handled by any visitor to the `variant`.
- All members of `variant` satisfy at least the basic guarantee of exception−safety. That is, all operations on a `variant` remain defined even after previous operations have failed.
- Each type specified as a template argument to `variant` must meet the requirements of the *BoundedType* concept.
- Each type specified as a template argument to `variant` must be distinct after removal of qualifiers. Thus, for instance, both `variant<int, int>` and `variant<int, const int>` have undefined behavior.
- Conforming implementations of `variant` must allow at least ten types as template arguments. The exact number of allowed arguments is exposed by the preprocessor macro `BOOST_VARIANT_LIMIT_TYPES`. (See `make_variant_over` for a means to specify the bounded types of a `variant` by the elements of an MPL or compatible Sequence, thus overcoming this limitation.)

`variant` construct/copy/destruct

1.
```
variant();
```

   Requires
   : The first bounded type of the `variant` (i.e., `T1`) must fulfill the requirements of the *DefaultConstructible* [20.1.4] concept.

   Postconditions
   : Content of `*this` is the default value of the first bounded type (i.e, `T1`).

   Throws
   : May fail with any exceptions arising from the default constructor of `T1`.

2.
```
variant(const variant & other);
```

   Postconditions
   : Content of `*this` is a copy of the content of `other`.

   Throws
   : May fail with any exceptions arising from the copy constructor of `other`'s contained type.

3.
```
template<typename T> variant(T & operand);
```

   Requires
   : `T` must be unambiguously convertible to one of the bounded types (i.e., `T1`, `T2`, etc.).

   Postconditions
   : Content of `*this` is the best conversion of `operand` to one of the bounded types, as determined by standard overload resolution rules.

   Throws
   : May fail with any exceptions arising from the conversion of `operand` to one of the bounded types.

4.
```
template<typename T> variant(const T & operand);
```

   Notes
   : Same semantics as previous constructor, but allows construction from temporaries.

5.
```
template<typename U1, typename U2, ..., typename UN> 
  variant(variant<U1, U2, ..., UN> & operand);
```

   Requires
   : *Every* one of `U1`, `U2`, ..., `UN` must have an unambiguous conversion to one of the bounded types (i.e., `T1`, `T2`, ..., `TN`).

   Postconditions
   : If `variant<U1, U2, ..., UN>` is itself one of the bounded types, then content of `*this` is a copy of `operand`. Otherwise, content of `*this` is the best conversion of the content of `operand` to one of the bounded types, as determined by standard overload resolution rules.

   Throws
   : If `variant<U1, U2, ..., UN>` is itself one of the bounded types, then may fail with any exceptions arising from the copy constructor of `variant<U1, U2, ..., UN>`. Otherwise, may fail with any exceptions arising from the conversion of the content of `operand` to one of the bounded types.

6.
```
template<typename U1, typename U2, ..., typename UN> 
  variant(const variant<U1, U2, ..., UN> & operand);
```

   Notes
   : Same semantics as previous constructor, but allows construction from temporaries.

7.
```
~variant();
```

   Effects
   : Destroys the content of `*this`.

   Throws
   : Will not throw.

`variant` modifiers

1. **`void`** `swap(variant & other);`

   Requires
   : Every bounded type must fulfill the requirements of the Assignable concept.

   Effects
   : Interchanges the content of `*this` and `other`.

   Throws
   : If the contained type of `other` is the same as the contained type of `*this`, then may fail with any exceptions arising from the `swap` of the contents of `*this` and `other`. Otherwise, may fail with any exceptions arising from either of the copy constructors of the contained types. Also, in the event of insufficient memory, may fail with `std::bad_alloc` (why?).

2. `variant &` **`operator`**`=(`**`const`** `variant & rhs);`

   Requires
   : Every bounded type must fulfill the requirements of the Assignable concept.

   Effects
   : If the contained type of `rhs` is the same as the contained type of `*this`, then assigns the content of `rhs` into the content of `*this`. Otherwise, makes the content of `*this` a copy of the content of `rhs`, destroying the previous content of `*this`.

   Throws
   : If the contained type of `rhs` is the same as the contained type of `*this`, then may fail with any exceptions arising from the assignment of the content of `rhs` into the content `*this`. Otherwise, may fail with any exceptions arising from the copy constructor of the contained type of `rhs`. Also, in the event of insufficient memory, may fail with `std::bad_alloc` (why?).

3. **`template`**`<`**`typename`** `T> variant &` **`operator`**`=(`**`const`** `T & rhs);`

   Requires
   : - `T` must be unambiguously convertible to one of the bounded types (i.e., `T1`, `T2`, etc.).
     - Every bounded type must fulfill the requirements of the Assignable concept.

   Effects
   : If the contained type of `*this` is `T`, then assigns `rhs` into the content of `*this`. Otherwise, makes the content of `*this` the best conversion of `rhs` to one of the bounded types, as determined by standard overload resolution rules, destroying the previous content of `*this`.

   Throws
   : If the contained type of `*this` is `T`, then may fail with any exceptions arising from the assignment of `rhs` into the content `*this`. Otherwise, may fail with any exceptions arising from the conversion of `rhs` to one of the bounded types. Also, in the event of insufficient memory, may fail with `std::bad_alloc` (why?).

`variant` queries

1. **`int`** `which()` **`const`**`;`

   Returns
   : The zero−based index into the set of bounded types of the contained type of `*this`. (For instance, if called on a `variant<int, std::string>` object containing a `std::string`, `which()` would return `1`.)

   Throws
   : Will not throw.

2. **`bool`** `empty()` **`const`**`;`

   Returns
   : `false`: `variant` always contains exactly one of its bounded types. (See Section 4.1, "Never−Empty" Guarantee for more information.)

   Rationale

Facilitates generic compatibility with boost::any.

Throws

Will not throw.

3. `const std::type_info & type() const;`

Returns

`typeid(x)`, where `x` is the the content of `*this`.

Throws

Will not throw.

`variant` relational

1.
```
bool operator==(const variant & rhs) const;
template<typename U> void operator==(const U & ) const;
```

Notes

The overload returning `void` exists only to prohibit implicit conversion of the operator's right–hand side to `variant`; thus, its use will (purposefully) result in a compile–time error.

Requires

Every bounded type of the `variant` must fulfill the requirements of the EqualityComparable concept.

Returns

`true` iff `which() == rhs.which()` *and* `content_this == content_rhs`, where `content_this` is the content of `*this` and `content_rhs` is the content of `rhs`.

Throws

If `which() == rhs.which()` then may fail with any exceptions arising from `operator==(T,T)`, where `T` is the contained type of `*this`.

2.
```
bool operator<(const variant & rhs) const;
template<typename U> void operator<(const U & ) const;
```

Notes

The overload returning `void` exists only to prohibit implicit conversion of the operator's right–hand side to `variant`; thus, its use will (purposefully) result in a compile–time error.

Requires

Every bounded type of the `variant` must fulfill the requirements of the LessThanComparable concept.

Returns

If `which() == rhs.which()` then: `content_this < content_rhs`, where `content_this` is the content of `*this` and `content_rhs` is the content of `rhs`. Otherwise: `which() < rhs.which()`.

Throws

If `which() == rhs.which()` then may fail with any exceptions arising from `operator<(T,T)`, where `T` is the contained type of `*this`.

Function template swap

boost::swap

Synopsis

```
template<typename T1, typename T2, ..., typename TN>
  void swap(variant<T1, T2, ..., TN> & lhs, variant<T1, T2, ..., TN> & rhs);
```

Description

Effects
    Swaps `lhs` with `rhs` by application of `variant::swap`.
Throws
    May fail with any exception arising from `variant::swap`.

Function template operator<<

boost::operator<<   Provides streaming output for `variant` types.

Synopsis

```
template<typename ElemType, typename Traits, typename T1, typename T2, ...,
         typename TN>
  std::basic_ostream<ElemType,Traits> &
  operator<<(std::basic_ostream<ElemType,Traits> & out,
             const variant<T1, T2, ..., TN> & rhs);
```

Description

Requires
    Every bounded type of the `variant` must fulfill the requirements of the *OutputStreamable* concept.
Effects
    Calls `out << x`, where `x` is the content of `rhs`.

Class template make_variant_over

boost::make_variant_over

Exposes a `variant` whose bounded types are the elements of the given type sequence.

Synopsis

```
template<typename Sequence>
class make_variant_over {
public:
  // types
  typedef variant< unspecified > type;
};
```

Description

`type` has behavior equivalent in every respect to `variant< Sequence[0], Sequence[1], ... >` (where `Sequence[i]` denotes the *i*–th element of `Sequence`), except that no upper limit is imposed on the number of types.

Notes:

- `Sequence` must meet the requirements of MPL's *Sequence* concept.
- Due to standard conformance problems in several compilers, `make_variant_over` may not be supported on your compiler. See `BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT` for more information.

3.5. Header <boost/variant/recursive_variant.hpp>

```
namespace boost {
  typedef unspecified recursive_variant_;

  template<typename T1, typename T2 = unspecified, ...,
           typename TN = unspecified>
    class make_recursive_variant;
  template<typename Sequence> class make_recursive_variant_over;
}
```

Class template make_recursive_variant

boost::make_recursive_variant Simplifies declaration of recursive `variant` types.

Synopsis

```
template<typename T1, typename T2 = unspecified, ...,
         typename TN = unspecified>
class make_recursive_variant {
public:
  // types
  typedef boost::variant< unspecified > type;
};
```

Description

`type` has behavior equivalent in every respect to some `variant< U1, U2, ..., UN >`, where each type `Ui` is the result of the corresponding type `Ti` undergone a transformation function. The following pseudo–code specifies the behavior of this transformation (call it `substitute`):

- If `Ti` is `boost::recursive_variant_` then: `variant< U1, U2, ..., UN >`;
- Else if `Ti` is of the form `X *` then: `substitute(X) *`;
- Else if `Ti` is of the form `X &` then: `substitute(X) &`;
- Else if `Ti` is of the form `R (*)( X1, X2, ..., XN )` then: `substitute(R) (*)( substitute(X1), substitute(X2), ..., substitute(XN) )`;
- Else if `Ti` is of the form `F < X1, X2, ..., XN >` then: `F< substitute(X1), substitute(X2), ..., substitute(XN) >`;
- Else: `Ti`.

Note that cv–qualifiers are preserved and that the actual process is generally a bit more complicated. However, the above does convey the essential idea as well as describe the extent of the substititions.

Use of `make_recursive_variant` is demonstrated in Section 2.2.3.2, Recursive types with `make_recursive_variant` .

Portability: Due to standard conformance issues in several compilers, `make_recursive_variant` is not universally supported. On these compilers the library indicates its lack of support via the definition of the preprocessor symbol `BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT`.

Class template make_recursive_variant_over

boost::make_recursive_variant_over

Exposes a recursive `variant` whose bounded types are the elements of the given type sequence.

Synopsis

```
template<typename Sequence>
class make_recursive_variant_over {
public:
  // types
  typedef variant< unspecified > type;
};
```

Description

`type` has behavior equivalent in every respect to `make_recursive_variant< Sequence[0], Sequence[1], ... >::type` (where `Sequence[i]` denotes the *i*–th element of `Sequence`), except that no upper limit is imposed on the number of types.

Notes:

- `Sequence` must meet the requirements of MPL's *Sequence* concept.
- Due to standard conformance problems in several compilers, `make_recursive_variant_over` may not be supported on your compiler. See `BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT` for more information.

3.6. Header <boost/variant/recursive_wrapper.hpp>

```
namespace boost {
  template<typename T> class recursive_wrapper;
  template<typename T> class is_recursive_wrapper;
  template<typename T> class unwrap_recursive_wrapper;
}
```

Class template recursive_wrapper

boost::recursive_wrapper   Solves circular dependencies, enabling recursive types.

Synopsis

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

  // construct/copy/destruct
  recursive_wrapper();
  recursive_wrapper(const recursive_wrapper &);
  recursive_wrapper(const T &);
  ~recursive_wrapper();

  // modifiers
  void swap(recursive_wrapper &);
  recursive_wrapper & operator=(const recursive_wrapper &);
  recursive_wrapper & operator=(const T &);

  // queries
  T & get();
  const T & get() const;
  T * get_pointer();
  const T * get_pointer() const;
};
```

Description

The `recursive_wrapper` class template has an interface similar to a simple value container, but its content is allocated dynamically. This allows `recursive_wrapper` to hold types `T` whose member data leads to a circular dependency (e.g., a data member of `T` has a data member of type `T`).

The application of `recursive_wrapper` is easiest understood in context. See Section 2.2.3.1, Recursive types with `recursive_wrapper` for a demonstration of a common use of the class template.

Notes:

- Any type specified as the template argument to `recursive_wrapper` must be capable of construction via `operator new`. Thus, for instance, references are not supported.

`recursive_wrapper` construct/copy/destruct

1. `recursive_wrapper();`

   Initializes `*this` by default construction of `T`.

   Requires
   : `T` must fulfill the requirements of the *DefaultConstructible* [20.1.4] concept.

   Throws
   : May fail with any exceptions arising from the default constructor of `T` or, in the event of insufficient memory, with `std::bad_alloc`.
2. `recursive_wrapper(const recursive_wrapper & other);`

   Copies the content of `other` into `*this`.

   Throws
   : May fail with any exceptions arising from the copy constructor of `T` or, in the event of insufficient memory, with `std::bad_alloc`.
3. `recursive_wrapper(const T & operand);`

   Copies `operand` into `*this`.

   Throws
   : May fail with any exceptions arising from the copy constructor of `T` or, in the event of insufficient memory, with `std::bad_alloc`.
4. `~recursive_wrapper();`

   Deletes the content of `*this`.

   Throws
   : Will not throw.

`recursive_wrapper` modifiers

1. `void swap(recursive_wrapper & other);`

   Exchanges contents of `*this` and `other`.

   Throws
   : Will not throw.
2. `recursive_wrapper & operator=(const recursive_wrapper & rhs);`

Assigns the content of `rhs` to the content of `*this`.

Requires
: `T` must fulfill the requirements of the Assignable concept.

Throws
: May fail with any exceptions arising from the assignment operator of `T`.

3. `recursive_wrapper & operator=(const T & rhs);`

   Assigns `rhs` into the content of `*this`.

   Requires
   : `T` must fulfill the requirements of the Assignable concept.

   Throws
   : May fail with any exceptions arising from the assignment operator of `T`.

`recursive_wrapper` queries

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

   Returns a reference to the content of `*this`.

   Throws
   : Will not throw.

2.
   ```
   T * get_pointer();
   const T * get_pointer() const;
   ```

   Returns a pointer to the content of `*this`.

   Throws
   : Will not throw.

Class template is_recursive_wrapper

boost::is_recursive_wrapper

Determines whether the specified type is a specialization of `recursive_wrapper`.

Synopsis

```
template<typename T>
class is_recursive_wrapper {
public:
  // types
  typedef unspecified type;

  // static constants
  static const bool value = unspecified;
};
```

Description

Value is true iff `T` is a specialization of `recursive_wrapper`.

Note: `is_recursive_wrapper` is a model of MPL's *IntegralConstant* concept.

Class template unwrap_recursive_wrapper

boost::unwrap_recursive_wrapper

Unwraps the specified argument if given a specialization of `recursive_wrapper`.

Synopsis

```
template<typename T>
class unwrap_recursive_wrapper {
public:
  // types
  typedef unspecified type;
};
```

Description

`type` is equivalent to `T::type` if `T` is a specialization of `recursive_wrapper`. Otherwise, `type` is equivalent to `T`.

3.7. Header <boost/variant/apply_visitor.hpp>

```
namespace boost {
  template<typename Visitor> class apply_visitor_delayed_t;
  template<typename Visitor, typename Variant>
    typename Visitor::result_type apply_visitor(Visitor &, Variant &);
  template<typename Visitor, typename Variant>
    typename Visitor::result_type apply_visitor(const Visitor &, Variant &);
  template<typename BinaryVisitor, typename Variant1, typename Variant2>
    typename BinaryVisitor::result_type
    apply_visitor(BinaryVisitor &, Variant1 &, Variant2 &);
  template<typename BinaryVisitor, typename Variant1, typename Variant2>
    typename BinaryVisitor::result_type
    apply_visitor(const BinaryVisitor &, Variant1 &, Variant2 &);
  template<typename Visitor>
    apply_visitor_delayed_t<Visitor> apply_visitor(Visitor &);
}
```

Class template apply_visitor_delayed_t

boost::apply_visitor_delayed_t — Adapts a visitor for use as a function object.

Synopsis

```
template<typename Visitor>
class apply_visitor_delayed_t {
public:
  // types
  typedef typename Visitor::result_type result_type;

  // construct/copy/destruct
  explicit apply_visitor_delayed_t(Visitor &);

  // function object interface
  template<typename Variant> result_type operator()(Variant &);
  template<typename Variant1, typename Variant2>
    result_type operator()(Variant1 &, Variant2 &);
};
```

Description

Adapts the function given at construction for use as a function object. This is useful, for example, when one needs to operate on each element of a sequence of variant objects using a standard library algorithm such as `std::for_each`.

See the "visitor−only" form of `apply_visitor` for a simple way to create `apply_visitor_delayed_t` objects.

`apply_visitor_delayed_t` construct/copy/destruct

1.
   ```
   explicit apply_visitor_delayed_t(Visitor & visitor);
   ```

   Effects

   Constructs the function object with the given visitor.

`apply_visitor_delayed_t` function object interface

1.
   ```
   template<typename Variant> result_type operator()(Variant & operand);
   template<typename Variant1, typename Variant2>
     result_type operator()(Variant1 & operand1, Variant2 & operand2);
   ```

   Invokes `apply_visitor` on the stored visitor using the given operands.

Function apply_visitor

boost::apply_visitor

Allows compile−time checked type−safe application of the given visitor to the content of the given variant, ensuring that all types are handled by the visitor.

Synopsis

```
template<typename Visitor, typename Variant>
  typename Visitor::result_type
  apply_visitor(Visitor & visitor, Variant & operand);
template<typename Visitor, typename Variant>
  typename Visitor::result_type
  apply_visitor(const Visitor & visitor, Variant & operand);
template<typename BinaryVisitor, typename Variant1, typename Variant2>
  typename BinaryVisitor::result_type
  apply_visitor(BinaryVisitor & visitor, Variant1 & operand1,
                Variant2 & operand2);
template<typename BinaryVisitor, typename Variant1, typename Variant2>
  typename BinaryVisitor::result_type
  apply_visitor(const BinaryVisitor & visitor, Variant1 & operand1,
                Variant2 & operand2);
template<typename Visitor>
  apply_visitor_delayed_t<Visitor> apply_visitor(Visitor & visitor);
```

Description

The behavior of `apply_visitor` is dependent on the number of arguments on which it operates (i.e., other than the visitor). The function behaves as follows:

- Overloads accepting one operand invoke the unary function call operator of the given visitor on the content of the given `variant` operand.

- Overloads accepting two operands invoke the binary function call operator of the given visitor on the content of the given `variant` operands.
- The overload accepting only a visitor returns a generic function object that accepts either one or two arguments and invokes `apply_visitor` using these arguments and `visitor`, thus behaving as specified above. (This behavior is particularly useful, for example, when one needs to operate on each element of a sequence of variant objects using a standard library algorithm.)

Returns
: The overloads acccepting operands return the result of applying the given visitor to the content of the given operands. The overload accepting only a visitor return a function object, thus delaying application of the visitor to any operands.

Requires
: The given visitor must fulfill the *StaticVisitor* concept requirements with respect to each of the bounded types of the given `variant`.

Throws
: The overloads accepting operands throw only if the given visitor throws when applied. The overload accepting only a visitor will not throw. (Note, however, that the returned function object may throw when invoked.)

3.8. Header <boost/variant/get.hpp>

```
namespace boost {
  class bad_get;
  template<typename U, typename T1, typename T2, ..., typename TN>
    U * get(variant<T1, T2, ..., TN> *);
  template<typename U, typename T1, typename T2, ..., typename TN>
    const U * get(const variant<T1, T2, ..., TN> *);
  template<typename U, typename T1, typename T2, ..., typename TN>
    U & get(variant<T1, T2, ..., TN> &);
  template<typename U, typename T1, typename T2, ..., typename TN>
    const U & get(const variant<T1, T2, ..., TN> &);
}
```

Class bad_get

boost::bad_get

The exception thrown in the event of a failed application of `boost::get` on the given operand value.

Synopsis

```
class bad_get : public std::exception {
public:
  virtual constchar * what() const;
};
```

Description

```
virtual constchar * what() const;
```

Function get

boost::get

Retrieves a value of a specified type from a given `variant`.

Synopsis

```
template<typename U, typename T1, typename T2, ..., typename TN>
  U * get(variant<T1, T2, ..., TN> * operand);
template<typename U, typename T1, typename T2, ..., typename TN>
  const U * get(const variant<T1, T2, ..., TN> * operand);
template<typename U, typename T1, typename T2, ..., typename TN>
  U & get(variant<T1, T2, ..., TN> & operand);
template<typename U, typename T1, typename T2, ..., typename TN>
  const U & get(const variant<T1, T2, ..., TN> & operand);
```

Description

The `get` function allows run–time checked, type–safe retrieval of the content of the given `variant`. The function succeeds only if the content is of the specified type `U`, with failure indicated as described below.

Warning: After either `operand` or its content is destroyed (e.g., when the given `variant` is assigned a value of different type), the returned reference is invalidated. Thus, significant care and caution must be extended when handling the returned reference.

Notes
: As part of its guarantee of type–safety, `get` enforces `const`–correctness. Thus, the specified type `U` must be `const`–qualified whenever `operand` or its content is likewise `const`–qualified. The converse, however, is not required: that is, the specified type `U` may be `const`–qualified even when `operand` and its content are not.

Returns
: If passed a pointer, `get` returns a pointer to the value content if it is of the specified type `U`; otherwise, a null pointer is returned. If passed a reference, `get` returns a reference to the value content if it is of the specified type `U`; otherwise, an exception is thrown (see below).

Throws
: Overloads taking a `variant` pointer will not throw; the overloads taking a `variant` reference throw `bad_get` if the content is not of the specified type `U`.

Rationale
: While visitation via `apply_visitor` is generally prefered due to its greater safety, `get` may may be more convenient in some cases due to its straightforward usage.

3.9. Header <boost/variant/bad_visit.hpp>

```
namespace boost {
  class bad_visit;
}
```

Class bad_visit

boost::bad_visit

The exception thrown in the event of a visitor unable to handle the visited value.

Synopsis

```
class bad_visit : public std::exception {
public:
  virtual const char * what() const;
};
```

Description

```
virtual const char * what() const;
```

3.10. Header <boost/variant/static_visitor.hpp>

```
namespace boost {
  template<typename ResultType> class static_visitor;
}
```

Class template static_visitor

boost::static_visitor — Convenient base type for static visitors.

Synopsis

```
template<typename ResultType>
class static_visitor {
public:
  // types
  typedef ResultType result_type;  // Exposes result_type member as required by StaticVisitor concept.
};
```

Description

Denotes the intent of the deriving class as meeting the requirements of a static visitor of some type. Also exposes the inner type `result_type` as required by the *StaticVisitor* concept.

Notes: `static_visitor` is intended for use as a base type only and is therefore noninstantiable.

3.11. Header <boost/variant/visitor_ptr.hpp>

```
namespace boost {
  template<typename T, typename R> class visitor_ptr_t;
  template<typename R, typename T> visitor_ptr_t<T,R> visitor_ptr(R (*)(T));
}
```

Class template visitor_ptr_t

boost::visitor_ptr_t — Adapts a function pointer for use as a static visitor.

Synopsis

```
template<typename T, typename R>
class visitor_ptr_t : public static_visitor<R> {
public:
  // construct/copy/destruct
  explicit visitor_ptr_t(R (*)(T));

  // static visitor interfaces
  R operator()(unspecified-forwarding-type);
  template<typename U> void operator()(const U&);
};
```

Description

Adapts the function given at construction for use as a static visitor of type `T` with result type `R`.

`visitor_ptr_t` construct/copy/destruct

1.
```
explicit visitor_ptr_t(R (*)(T) );
```

   Effects
      Constructs the visitor with the given function.

`visitor_ptr_t` static visitor interfaces

1.
```
R operator()(unspecified-forwarding-type operand);
template<typename U> void operator()(const U& );
```

   Effects
      If passed a value or reference of type `T`, it invokes the function given at construction, appropriately forwarding `operand`.

   Returns
      Returns the result of the function invocation.

   Throws
      The overload taking a value or reference of type `T` throws if the invoked function throws. The overload taking all other values *always* throws `bad_visit`.

Function template visitor_ptr

boost::visitor_ptr

Returns a visitor object that adapts function pointers for use as a static visitor.

Synopsis

```
template<typename R, typename T> visitor_ptr_t<T,R> visitor_ptr(R (*)(T) );
```

Description

Constructs and returns a `visitor_ptr_t` adaptor over the given function.

Returns
   Returns a `visitor_ptr_t` visitor object that, when applied, invokes the given function.

Throws
   Will not throw. (Note, however, that the returned visitor object may throw when applied.)

4. Design Overview

4.1. "Never-Empty" Guarantee

4.1.1. The Guarantee

All instances `v` of type `variant<T1,T2,...,TN>` guarantee that `v` has constructed content of one of the types `Ti`, even if an operation on `v` has previously failed.

This implies that `variant` may be viewed precisely as a union of *exactly* its bounded types. This "never−empty" property insulates the user from the possibility of undefined `variant` content and the significant additional complexity−of−use attendant with such a possibility.

4.1.2. The Implementation Problem

While the never−empty guarantee might at first seem "obvious," it is in fact not even straightforward how to implement it in general (i.e., without unreasonably restrictive additional requirements on bounded types).

The central difficulty emerges in the details of `variant` assignment. Given two instances `v1` and `v2` of some concrete `variant` type, there are two distinct, fundamental cases we must consider for the assignment `v1 = v2`.

First consider the case that `v1` and `v2` each contains a value of the same type. Call this type `T`. In this situation, assignment is perfectly straightforward: use `T::operator=`.

However, we must also consider the case that `v1` and `v2` contain values *of distinct types*. Call these types `T` and `U`. At this point, since `variant` manages its content on the stack, the left−hand side of the assignment (i.e., `v1`) must destroy its content so as to permit in−place copy−construction of the content of the right−hand side (i.e., `v2`). In the end, whereas `v1` began with content of type `T`, it ends with content of type `U`, namely a copy of the content of `v2`.

The crux of the problem, then, is this: in the event that copy−construction of the content of `v2` fails, how can `v1` maintain its "never−empty" guarantee? By the time copy−construction from `v2` is attempted, `v1` has already destroyed its content!

4.1.3. The "Ideal" Solution: False Hopes

Upon learning of this dilemma, clever individuals may propose the following scheme hoping to solve the problem:

1. Provide some "backup" storage, appropriately aligned, capable of holding values of the contained type of the left−hand side.
2. Copy the memory (e.g., using `memcpy`) of the storage of the left−hand side to the backup storage.
3. Attempt a copy of the right−hand side content to the (now−replicated) left−hand side storage.
4. In the event of an exception from the copy, restore the backup (i.e., copy the memory from the backup storage back into the left−hand side storage).
5. Otherwise, in the event of success, now copy the memory of the left−hand side storage to another "temporary" aligned storage.
6. Now restore the backup (i.e., again copying the memory) to the left−hand side storage; with the "old" content now restored, invoke the destructor of the contained type on the storage of the left−hand side.
7. Finally, copy the memory of the temporary storage to the (now−empty) storage of the left−hand side.

While complicated, it appears such a scheme could provide the desired safety in a relatively efficient manner. In fact, several early iterations of the library implemented this very approach.

Unfortunately, as Dave Abraham's first noted, the scheme results in undefined behavior:

> "That's a lot of code to read through, but if it's doing what I think it's doing, it's undefined behavior.
>
> "Is the trick to move the bits for an existing object into a buffer so we can tentatively construct a new object in that memory, and later move the old bits back temporarily to destroy the old object?
>
> "The standard does not give license to do that: only one object may have a given address at a time. See 3.8, and particularly paragraph 4."

Additionally, as close examination quickly reveals, the scheme has the potential to create irreconcilable race−conditions in concurrent environments.

Ultimately, even if the above scheme could be made to work on certain platforms with particular compilers, it is still necessary to find a portable solution.

4.1.4. An Initial Solution: Double Storage

Upon learning of the infeasibility of the above scheme, Anthony Williams proposed in [Wil02] a scheme that served as the basis for a portable solution in some pre–release implementations of `variant`.

The essential idea to this scheme, which shall be referred to as the "double storage" scheme, is to provide enough space within a `variant` to hold two separate values of any of the bounded types.

With the secondary storage, a copy the right–hand side can be attempted without first destroying the content of the left–hand side; accordingly, the content of the left–hand side remains available in the event of an exception.

Thus, with this scheme, the `variant` implementation needs only to keep track of which storage contains the content –– and dispatch any visitation requests, queries, etc. accordingly.

The most obvious flaw to this approach is the space overhead incurred. Though some optimizations could be applied in special cases to eliminate the need for double storage –– for certain bounded types or in some cases entirely (see Section 4.1.6, Enabling Optimizations for more details) –– many users on the Boost mailing list strongly objected to the use of double storage. In particular, it was noted that the overhead of double storage would be at play at all times –– even if assignment to `variant` never occurred. For this reason and others, a new approach was developed.

4.1.5. Current Approach: Temporary Heap Backup

Despite the many objections to the double storage solution, it was realized that no replacement would be without drawbacks. Thus, a compromise was desired.

To this end, Dave Abrahams suggested to include the following in the behavior specification for `variant` assignment: "`variant` assignment from one type to another may incur dynamic allocation." That is, while `variant` would continue to store its content *in situ* after construction and after assignment involving identical contained types, `variant` would store its content on the heap after assignment involving distinct contained types.

The algorithm for assignment would proceed as follows:

1. Copy–construct the content of the right–hand side to the heap; call the pointer to this data `p`.
2. Destroy the content of the left–hand side.
3. Copy `p` to the left–hand side storage.

Since all operations on pointers are nothrow, this scheme would allow `variant` to meet its never–empty guarantee.

The most obvious concern with this approach is that while it certainly eliminates the space overhead of double storage, it introduces the overhead of dynamic–allocation to `variant` assignment –– not just in terms of the initial allocation but also as a result of the continued storage of the content on the heap. While the former problem is unavoidable, the latter problem may be avoided with the following "temporary heap backup" technique:

1. Copy–construct the content of the *left*–hand side to the heap; call the pointer to this data `backup`.
2. Destroy the content of the left–hand side.
3. Copy–construct the content of the right–hand side in the (now–empty) storage of the left–hand side.
4. In the event of failure, copy `backup` to the left–hand side storage.
5. In the event of success, deallocate the data pointed to by `backup`.

With this technique: 1) only a single storage is used; 2) allocation is on the heap in the long–term only if the assignment fails; and 3) after any *successful* assignment, storage within the `variant` is guaranteed. For the purposes of the initial release of the library, these characteristics were deemed a satisfactory compromise solution.

There remain notable shortcomings, however. In particular, there may be some users for which heap allocation must be avoided at all costs; for other users, any allocation may need to occur via a user–supplied allocator. These issues will be addressed in the future (see Section 4.1.7, Future Direction: Policy–based Implementation). For now, though, the library treats storage of its content as an implementation detail. Nonetheless, as described in the next section, there *are* certain things the user can do to ensure the greatest efficiency for `variant` instances (see Section 4.1.6, Enabling Optimizations for details).

4.1.6. Enabling Optimizations

As described in Section 4.1.2, The Implementation Problem , the central difficulty in implementing the never−empty guarantee is the possibility of failed copy−construction during `variant` assignment. Yet types with nothrow copy constructors clearly never face this possibility. Similarly, if one of the bounded types of the `variant` is nothrow default−constructible, then such a type could be used as a safe "fallback" type in the event of failed copy construction.

Accordingly, `variant` is designed to enable the following optimizations once the following criteria on its bounded types are met:

- For each bounded type `T` that is nothrow copy−constructible (as indicated by `boost::has_nothrow_copy`), the library guarantees `variant` will use only single storage and in−place construction for `T`.
- If *any* bounded type is nothrow default−constructible (as indicated by `boost::has_nothrow_constructor`), the library guarantees `variant` will use only single storage and in−place construction for *every* bounded type in the `variant`. Note, however, that in the event of assignment failure, an unspecified nothrow default−constructible bounded type will be default−constructed in the left−hand side operand so as to preserve the never−empty guarantee.

Caveat: On most platforms, the Type Traits templates `has_nothrow_copy` and `has_nothrow_constructor` by default return `false` for all `class` and `struct` types. It is necessary therefore to provide specializations of these templates as appropriate for user−defined types, as demonstrated in the following:

```
// ...in your code (at file scope)...

namespace boost {

  template <>
  struct has_nothrow_copy< myUDT >
    : mpl::true_
  {
  };

}
```

Implementation Note: So as to make the behavior of `variant` more predictable in the aftermath of an exception, the current implementation prefers to default−construct `boost::blank` if specified as a bounded type instead of other nothrow default−constructible bounded types. (If this is deemed to be a useful feature, it will become part of the specification for `variant`; otherwise, it may be obsoleted. Please provide feedback to the Boost mailing list.)

4.1.7. Future Direction: Policy−based Implementation

As the previous sections have demonstrated, much effort has been expended in an attempt to provide a balance between performance, data size, and heap usage. Further, significant optimizations may be enabled in `variant` on the basis of certain traits of its bounded types.

However, there will be some users for whom the chosen compromise is unsatisfactory (e.g.: heap allocation must be avoided at all costs; if heap allocation is used, custom allocators must be used; etc.). For this reason, a future version of the library will support a policy−based implementation of `variant`. While this will not eliminate the problems described in the previous sections, it will allow the decisions regarding tradeoffs to be decided by the user rather than the library designers.

5. Miscellaneous Notes

5.1. Boost.Variant vs. Boost.Any

As a discriminated union container, the Variant library shares many of the same features of the Any library. However, since neither library wholly encapsulates the features of the other, one library cannot be generally recommended for use over the other.

That said, Boost.Variant has several advantages over Boost.Any, such as:

- Boost.Variant guarantees the type of its content is one of a finite, user–specified set of types.
- Boost.Variant provides *compile–time* checked visitation of its content. (By contrast, the current version of Boost.Any provides no visitation mechanism at all; but even if it did, it would need to be checked at run–time.)
- Boost.Variant enables generic visitation of its content. (Even if Boost.Any did provide a visitation mechanism, it would enable visitation only of explicitly–specified types.)
- Boost.Variant offers an efficient, stack–based storage scheme (avoiding the overhead of dynamic allocation).

Of course, Boost.Any has several advantages over Boost.Variant, such as:

- Boost.Any, as its name implies, allows virtually any type for its content, providing great flexibility.
- Boost.Any provides the no–throw guarantee of exception safety for its swap operation.
- Boost.Any makes little use of template metaprogramming techniques (avoiding potentially hard–to–read error messages and significant compile–time processor and memory demands).

5.2. Portability

The library aims for 100% ANSI/ISO C++ conformance. However, this is strictly impossible due to the inherently non–portable nature of the Type Traits library's `type_with_alignment` facility. In practice though, no compilers or platforms have been discovered where this reliance on undefined behavior has been an issue.

Additionally, significant effort has been expended to ensure proper functioning despite various compiler bugs and other conformance problems. To date the library testsuite has been compiled and tested successfully on at least the following compilers for basic and advanced functionality:

	Basic	**`variant<T&>`**	**`make_variant_over`**	**`make_recursive_variant`**
Borland C++ 5.5.1 and 5.6.4	X	X		
Comeau C++ 4.3.0	X	X	X	X
GNU GCC 3.3.1	X	X	X	X
GNU GCC 2.95.3	X	X		X
Intel C++ 7.0	X		X	X
Metrowerks CodeWarrior 8.3	X		X	X
Microsoft Visual C++ 7.1	X	X	X	X
Microsoft Visual C++ 6 SP5 and 7	X			

Finally, the current state of the testsuite in CVS may be found on the Test Summary page. Please note, however, that this page reports on day–to–day changes to inter–release code found in the Boost CVS and thus likely does not match the state of code found in Boost releases.

5.3. Troubleshooting

Due to the heavy use of templates in the implementation of `variant`, it is not uncommon when compiling to encounter problems related to template instantiaton depth, compiler memory, etc. This section attempts to provide advice to common problems experienced on several popular compilers.

(This section is still in progress, with additional advice/feedback welcome. Please post to the Boost−Users list with any useful experiences you may have.)

5.3.1. "Template instantiation depth exceeds maximum"

5.3.1.1. GNU GCC

The compiler option `-ftemplate-depth-`*`NN`* can increase the maximum allowed instantiation depth. (Try `-ftemplate-depth-50`.)

5.3.2. "Internal heap limit reached"

5.3.2.1. Microsoft Visual C++

The compiler option `/Zm`*`NNN`* can increase the memory allocation limit. The `NNN` is a scaling percentage (i.e., `100` denotes the default limit). (Try `/Zm200`.)

5.4. Acknowledgments

Eric Friedman and Itay Maman designed the initial submission; Eric was the primary implementer.

Eric is also the library maintainer and has expanded upon the initial submission −− adding `make_recursive_variant`, `make_variant_over`, support for reference content, etc.

Andrei Alexandrescu's work in [Ale01a] and [Ale02] inspired the library's design.

Jeff Garland was the formal review manager.

Douglas Gregor, Dave Abrahams, Anthony Williams, Fernando Cacciola, Joel de Guzman, Dirk Schreib, Brad King, Giovanni Bajo, Eugene Gladyshev, and others provided helpful feedback and suggestions to refine the semantics, interface, and implementation of the library.

6. References

[Abr00] David Abrahams. "Exception−Safety in Generic Components." M. Jazayeri, R. Loos, D. Musser (eds.): Generic Programming '98, Proc. of a Dagstuhl Seminar, Lecture Notes on Computer Science, Vol. 1766, pp. 69−79. Springer−Verlag Berlin Heidelberg. 2000.

[Abr01] David Abrahams. "Error and Exception Handling." Boost technical article. 2001−2003.

[Ale01a] Andrei Alexandrescu. "An Implementation of Discriminated Unions in C++." *OOPSLA 2001*, Second Workshop on C++ Template Programming. Tampa Bay, 14 October 2001.

[Ale01b] Andrei Alexandrescu. *Modern C++ Design*. Addison−Wesley, C++ In−Depth series. 2001.

[Ale02] Andrei Alexandrescu. "Generic<Programming>: Discriminated Unions" series: Part 1, Part 2, Part 3. *C/C++ Users Journal*. 2002.

[Boo02] Various Boost members. "Proposal −−− A type−safe union." Boost public discussion. 2002.

[C++98] *International Standard, Programming Languages C++*. ISO/IEC:14882. 1998.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object−Oriented Software*. Addison−Wesley. 1995.

[Gre02] Douglas Gregor. "BOOST_USER: variant." Boost Wiki paper. 2002.

MPL Aleksey Gurtovoy. *Boost Metaprogramming Library.* 2002.

[Hen01] Kevlin Henney. *Boost Any Library.* 2001.

Preprocessor Paul Mensonides and Vesa Karvonen. *Boost Preprocessor Library.* 2002.

Type Traits Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcus, John Maddock, Jeremy Siek. *Boost Type Traits Library*. 2001.

[Sut00] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison−Wesley, C++ In−Depth series. 2000.

[Wil02] Anthony Williams. Double−Storage Proposal. 2002.