Boost.Signals2

Douglas Gregor

Frank Mori Hess

Table of Contents

Introduction

The Boost.Signals2 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) can track 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.Signals2 allows the user to specify the manner in which multiple return values are combined.

Signals2

This documentation describes a thread-safe variant of the original Boost.Signals library. There have been some changes to the interface to support thread-safety, mostly with respect to automatic connection management. This implementation was written by Frank Mori Hess. Acknowledgements are also due to Timmo Stange, Peter Dimov, and Tony Van Eerd for ideas and feedback, and to Douglas Gregor for the original version of Boost.Signals this effort was based on.

Tutorial

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 Signals2 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 Signals2 library, that often require a solid working knowledge of the *Beginner* and *Intermediate* topics; most users will not need to read the *Advanced* sections.

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!".

```
struct HelloWorld
{
  void operator()() const
  {
    std::cout << "Hello, World!" << std::endl;
  }
};
```

```
  // Signal with no arguments and a void return value
  boost::signals2::signal<void ()> sig;

  // Connect a HelloWorld slot
  HelloWorld hello;
  sig.connect(hello);

  // Call all of the slots
  sig();
```

Calling Multiple Slots

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.

```
  boost::signals2::signal<void ()> sig;

  sig.connect(Hello());
  sig.connect(World());

  sig();
```

By default, slots are pushed onto the back of the slot list, so the output of this program will be as expected:

```
Hello, World!
```

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.Signals2 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:

```
  boost::signals2::signal<void ()> sig;

  sig.connect(1, World());  // connect with group 1
  sig.connect(0, Hello());  // connect with group 0
```

Invoking the signal 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::signals2::at_front` or `boost::signals2::at_back` as the last parameter to connect, respectively), and default to the end of the list. When a group is specified, the final `at_front` or `at_back` parameter describes where the slot will be placed within the group ordering. Ungrouped slots connected with `at_front` will always precede all grouped slots. Ungrouped slots connected with `at_back` will always succeed all grouped slots.

If we add a new slot to our example like this:

```
struct GoodMorning
{
  void operator()() const
  {
    std::cout << "... and good morning!" << std::endl;
  }
};
```

```
// by default slots are connected at the end of the slot list
sig.connect(GoodMorning());

// slots are invoked this order:
// 1) ungrouped slots connected with boost::signals2::at_front
// 2) grouped slots according to ordering of their groups
// 3) ungrouped slots connected with boost::signals2::at_back
sig();
```

... we will get the result we wanted:

```
Hello, World!
... and good morning!
```

Passing Values to and from Slots

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_args(float x, float y)
{
  std::cout << "The arguments are " << x << " and " << y << std::endl;
}

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;
}
```

```
  boost::signals2::signal<void (float, float)> sig;

  sig.connect(&print_args);
  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 arguments are 5 and 3
The sum is 8
The product is 15
The difference is 2
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::signals2::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.

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 may 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:

```
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::signals2::signal<float (float, float)> sig;
```

```
  sig.connect(&product);
  sig.connect(&quotient);
  sig.connect(&sum);
  sig.connect(&difference);

  // The default combiner returns a boost::optional containing the return
  // value of the last slot in the slot list, in this case the
  // difference function.
  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::signals2::signal` class template) is to call all slots and then return a `boost::optional` containing 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 connected.

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:

```
// combiner which returns the maximum value returned by all slots
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 floats). 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:

```
boost::signals2::signal<float (float x, float y),
               maximum<float> > sig;
```

Now we can connect slots that perform arithmetic functions and use the signal:

```
  sig.connect(&product);
  sig.connect(&quotient);
  sig.connect(&sum);
  sig.connect(&difference);

  // Outputs the maximum value returned by the connected slots, in this case
  // 15 from the product function.
  std::cout << "maximum: " << 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:

```
// aggregate_values is a combiner which places all the values returned
// from slots into a container
template<typename Container>
struct aggregate_values
{
  typedef Container result_type;

  template<typename InputIterator>
  Container operator()(InputIterator first, InputIterator last) const
  {
    Container values;

    while(first != last) {
      values.push_back(*first);
      ++first;
    }
    return values;
  }
};
```

Again, we can create a signal with this new combiner:

```
boost::signals2::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::cout << "aggregate values: ";
  std::copy(results.begin(), results.end(),
    std::ostream_iterator<float>(std::cout, " "));
  std::cout << "\n";
```

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;
  }
};
```

Connection Management

Disconnecting Slots (Beginner)

Slots aren't expected to exist indefinitely 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::signals2::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::signals2::connection c = sig.connect(HelloWorld());
std::cout << "c is connected\n";
sig(); // Prints "Hello, World!"

c.disconnect(); // Disconnect the HelloWorld object
std::cout << "c is disconnected\n";
sig(); // Does nothing: there are no connected slots
```

Blocking Slots (Beginner)

Slots can be temporarily "blocked", meaning that they will be ignored when the signal is invoked but have not been permanently disconnected. This is typically used to prevent infinite recursion in cases where otherwise running a slot would cause the signal it is connected to to be invoked again. A boost::signals2::shared_connection_block object will temporarily block a slot. The connection is unblocked by either destroying or calling unblock on all the shared_connection_block objects that reference the connection. Here is an example of blocking/unblocking slots:

```
boost::signals2::connection c = sig.connect(HelloWorld());
std::cout << "c is not blocked.\n";
sig(); // Prints "Hello, World!"

{
  boost::signals2::shared_connection_block block(c); // block the slot
  std::cout << "c is blocked.\n";
  sig(); // No output: the slot is blocked
} // shared_connection_block going out of scope unblocks the slot
std::cout << "c is not blocked.\n";
sig(); // Prints "Hello, World!"}
```

Scoped Connections (Intermediate)

The `boost::signals2::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::signals2::scoped_connection c(sig.connect(ShortLived()));
    sig(); // will call ShortLived function object
  } // scoped_connection goes out of scope and disconnects

  sig(); // ShortLived function object no longer connected to sig
```

Note, attempts to initialize a scoped_connection with the assignment syntax will fail due to it being noncopyable. Either the explicit initialization syntax or default construction followed by assignment from a `signals2::connection` will work:

```
// doesn't compile due to compiler attempting to copy a temporary scoped_connection object
// boost::signals2::scoped_connection c0 = sig.connect(ShortLived());

// okay
boost::signals2::scoped_connection c1(sig.connect(ShortLived()));

// also okay
boost::signals2::scoped_connection c2;
c2 = sig.connect(ShortLived());
```

Disconnecting Equivalent Slots (Intermediate)

One can disconnect slots that are equivalent to a given function object using a form of the `signal::disconnect` method, so long as the type of the function object has an accessible == operator. For instance:

```
void foo() { std::cout << "foo"; }
void bar() { std::cout << "bar\n"; }
```

```
boost::signals2::signal<void ()> sig;
```

```
  sig.connect(&foo);
  sig.connect(&bar);
  sig();

  // disconnects foo, but not bar
  sig.disconnect(&foo);
  sig();
```

Automatic Connection Management (Intermediate)

Boost.Signals2 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:

```
class NewsItem { /* ... */ };

typedef boost::signals2::signal<void (const NewsItem&)> signal_type;
signal_type 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.Signals2 one may track any object which is managed by a shared_ptr, by using `slot::track`. A slot will automatically disconnect when any of its tracked objects expire. In addition, Boost.Signals2 will ensure that no tracked object expires while the slot it is associated with is in mid-execution. It does so by creating temporary shared_ptr copies of the slot's tracked objects before executing it. To track `NewsMessageArea`, we use a shared_ptr to manage its lifetime, and pass the shared_ptr to the slot via its `slot::track` method before connecting it, e.g.:

```
// ...
boost::shared_ptr<NewsMessageArea> newsMessageArea(new NewsMessageArea(/* ... */));
// ...
deliverNews.connect(signal_type::slot_type(&NewsMessageArea::displayNews,
  newsMessageArea.get(), _1).track(newsMessageArea));
```

Note there is no explicit call to bind() needed in the above example. If the `signals2::slot` constructor is passed more than one argument, it will automatically pass all the arguments to `bind` and use the returned function object.

Also note, we pass an ordinary pointer as the second argument to the slot constructor, using `newsMessageArea.get()` instead of passing the `shared_ptr` itself. If we had passed the `newsMessageArea` itself, a copy of the `shared_ptr` would have been bound into the slot function, preventing the `shared_ptr` from expiring. However, the use of `slot::track` implies we wish to allow the tracked object to expire, and automatically disconnect the connection when this occurs.

`shared_ptr` classes other than `boost::shared_ptr` (such as `std::shared_ptr`) may also be tracked for connection management purposes. They are supported by the `slot::track_foreign` method.

Postconstructors and Predestructors (Advanced)

One limitation of using `shared_ptr` for tracking is that an object cannot setup tracking of itself in its constructor. However, it is possible to set up tracking in a post-constructor which is called after the object has been created and passed to a `shared_ptr`. The Boost.Signals2 library provides support for post-constructors and pre-destructors via the `deconstruct()` factory function.

For most cases, the simplest and most robust way to setup postconstructors for a class is to define an associated `adl_postconstruct` function which can be found by `deconstruct()`, make the class' constructors private, and give `deconstruct` access to the private constructors by declaring `deconstruct_access` a friend. This will ensure that objects of the class may only be created through the `deconstruct()` function, and their associated `adl_postconstruct()` function will always be called.

The examples section contains several examples of defining classes with postconstructors and predestructors, and creating objects of these classes using `deconstruct()`

Be aware that the postconstructor/predestructor support in Boost.Signals2 is in no way essential to the use of the library. The use of `deconstruct` is purely optional. One alternative is to define static factory functions for your classes. The factory function can create an object, pass ownership of the object to a `shared_ptr`, setup tracking for the object, then return the `shared_ptr`.

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.

• An object tracked by 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.

Note, even after a connection is disconnected, its's associated slot may still be in the process of executing. In other words, disconnection does not block waiting for the connection's associated slot to complete execution. This situation may occur in a multi-threaded environment if the disconnection occurs concurrently with signal invocation, or in a single-threaded environment if a slot disconnects itself.

Passing Slots (Intermediate)

Slots in the Boost.Signals2 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:

```
// a pretend GUI button
class Button
{
  typedef boost::signals2::signal<void (int x, int y)> OnClick;
public:
  typedef OnClick::slot_type OnClickSlotType;
  // forward slots through Button interface to its private signal
  boost::signals2::connection doOnClick(const OnClickSlotType & slot);

  // simulate user clicking on GUI button at coordinates 52, 38
  void simulateClick();
private:
  OnClick onClick;
};

boost::signals2::connection Button::doOnClick(const OnClickSlotType & slot)
{
  return onClick.connect(slot);
}

void Button::simulateClick()
{
  onClick(52, 38);
}

void printCoordinates(long x, long y)
{
  std::cout << "(" << x << ", " << y << ")\n";
}
```

```
  Button button;
  button.doOnClick(&printCoordinates);
  button.simulateClick();
```

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.

Example: Document-View

Signals can be used to implement flexible Document-View architectures. The document will contain a signal to which each of the views can connect. The following `Document` class defines a simple text document that supports mulitple views. Note that it stores a single signal to which all of the views will be connected.

```
class Document
{
public:
    typedef boost::signals2::signal<void ()>  signal_t;

public:
    Document()
    {}

    /* Connect a slot to the signal which will be emitted whenever
      text is appended to the document. */
    boost::signals2::connection connect(const signal_t::slot_type &subscriber)
    {
        return m_sig.connect(subscriber);
    }

    void append(const char* s)
    {
        m_text += s;
        m_sig();
    }

    const std::string& getText() const
    {
        return m_text;
    }

private:
    signal_t    m_sig;
    std::string m_text;
};
```

Next, we can begin to define views. The following `TextView` class provides a simple view of the document text.

```
class TextView
{
public:
    TextView(Document& doc): m_document(doc)
    {
        m_connection = m_document.connect(boost::bind(&TextView::refresh, this));
    }

    ~TextView()
    {
        m_connection.disconnect();
    }

    void refresh() const
    {
        std::cout << "TextView: " << m_document.getText() << std::endl;
    }
private:
    Document&                   m_document;
    boost::signals2::connection  m_connection;
};
```

Alternatively, we can provide a view of the document translated into hex values using the `HexView` view:

```
class HexView
{
public:
    HexView(Document& doc): m_document(doc)
    {
        m_connection = m_document.connect(boost::bind(&HexView::refresh, this));
    }

    ~HexView()
    {
        m_connection.disconnect();
    }

    void refresh() const
    {
        const std::string&  s = m_document.getText();

        std::cout << "HexView:";

        for (std::string::const_iterator it = s.begin(); it != s.end(); ++it)
            std::cout << ' ' << std::hex << static_cast<int>(*it);

        std::cout << std::endl;
    }
private:
    Document&               m_document;
    boost::signals2::connection  m_connection;
};
```

To tie the example together, here is a simple `main` function that sets up two views and then modifies the document:

```
int main(int argc, char* argv[])
{
    Document    doc;
    TextView    v1(doc);
    HexView     v2(doc);

    doc.append(argc == 2 ? argv[1] : "Hello world!");
    return 0;
}
```

The complete example source, contributed by Keith MacDonald, is available in the examples section. We also provide variations on the program which employ automatic connection management to disconnect views on their destruction.

Giving a Slot Access to its Connection (Advanced)

You may encounter situations where you wish to disconnect or block a slot's connection from within the slot itself. For example, suppose you have a group of asynchronous tasks, each of which emits a signal when it completes. You wish to connect a slot to all the tasks to retrieve their results as each completes. Once a given task completes and the slot is run, the slot no longer needs to be connected to the completed task. Therefore, you may wish to clean up old connections by having the slot disconnect its invoking connection when it runs.

For a slot to disconnect (or block) its invoking connection, it must have access to a `signals2::connection` object which references the invoking signal-slot connection. The difficulty is, the `connection` object is returned by the `signal::connect` method, and therefore is not available until after the slot is already connected to the signal. This can be particularly troublesome in a multi-threaded environment where the signal may be invoked concurrently by a different thread while the slot is being connected.

Therefore, the signal classes provide `signal::connect_extended` methods, which allow slots which take an extra argument to be connected to a signal. The extra argument is a `signals2::connection` object which refers to the signal-slot connection currently invoking the slot. `signal::connect_extended` uses slots of the type given by the `signal::extended_slot_type` typedef.

The examples section includes an extended_slot program which demonstrates the syntax for using signal::connect_extended.

Changing the `Mutex` Type of a Signal (Advanced).

For most cases the default type of boost::signals2::mutex for a signals2::signal's Mutex template type parameter should be fine. If you wish to use an alternate mutex type, it must be default-constructible and fulfill the Lockable concept defined by the Boost.Thread library. That is, it must have lock() and unlock() methods (the Lockable concept also includes a try_lock() method but this library does not require try locking).

The Boost.Signals2 library provides one alternate mutex class for use with signal: boost::signals2::dummy_mutex. This is a fake mutex for use in single-threaded programs, where locking a real mutex would be useless overhead. Other mutex types you could use with signal include boost::mutex, or the std::mutex from C++0x.

Changing a signal's Mutex template type parameter can be tedious, due to the large number of template parameters which precede it. The signal_type metafunction is particularly useful in this case, since it enables named template type parameters for the signals2::signal class. For example, to declare a signal which takes an int as an argument and uses a boost::signals2::dummy_mutex for its Mutex types, you could write:

```
namespace bs2 = boost::signals2;
using bs2::keywords;
bs2::signal_type<void (int), mutex_type<bs2::dummy_mutex> >::type sig;
```

Linking against the Signals2 library

Unlike the original Boost.Signals library, Boost.Signals2 is currently header-only.

Example programs

Miscellaneous Tutorial Examples

hello_world_slot

This example is a basic example of connecting a slot to a signal and then invoking the signal.

Download hello_world_slot.cpp.

hello_world_multi_slot

This example extends the hello_world_slot example slightly by connecting more than one slot to the signal before invoking it.

Download hello_world_multi_slot.cpp.

ordering_slots

This example extends the hello_world_multi_slot example slightly by using slot groups to specify the order slots should be invoked.

Download ordering_slots.cpp.

slot_arguments

The slot_arguments program shows how to pass arguments from a signal invocation to slots.

Download slot_arguments.cpp.

signal_return_value

This example shows how to return a value from slots to the signal invocation. It uses the default `optional_last_value` combiner.

Download signal_return_value.cpp.

custom_combiners

This example shows more returning of values from slots to the signal invocation. This time, custom combiners are defined and used.

Download custom_combiners.cpp.

disconnect_and_block

This example demonstrates various means of manually disconnecting slots, as well as temporarily blocking them via `shared_connection_block`.

Download disconnect_and_block.cpp.

passing_slots

This example demonstrates the passing of slot functions to a private signal through a non-template interface.

Download passing_slots.cpp.

extended_slot

This example demonstrates connecting an extended slot to a signal. An extended slot accepts a reference to its invoking signal-slot connection as an additional argument, permitting the slot to temporarily block or permanently disconnect itself.

Download extended_slot.cpp.

Document-View

doc_view

This is the document-view example program which is described in the tutorial. It shows usage of a signal and slots to implement two different views of a text document.

Download doc_view.cpp.

doc_view_acm

This program modifies the original doc_view.cpp example to employ automatic connection management.

Download doc_view_acm.cpp.

doc_view_acm_deconstruct

This program modifies the doc_view_acm.cpp example to use postconstructors and the `deconstruct()` factory function.

Download doc_view_acm_deconstruct.cpp.

Postconstructors and Predestructors with `deconstruct()`

postconstructor_ex1

This program is a basic example of how to define a class with a postconstructor which uses `deconstruct()` as its factory function.

Download postconstructor_ex1.

postconstructor_ex2

This program extends the postconstructor_ex1 example slightly, by additionally passing arguments from the `deconstruct()` call through to the class' constructor and postconstructor.

Download postconstructor_ex2.

predestructor_example

This program is a basic example of how to define a class with a predestructor which uses `deconstruct()` as its factory function.

Download predestructor_example.

Reference

Header <boost/signals2.hpp>

Including the "boost/signals2.hpp" header pulls in all the other headers of the Signals2 library. It is provided as a convenience.

Header <boost/signals2/connection.hpp>

```
namespace boost {
  namespace signals2 {
    class connection;
    void swap(connection&, connection&);
    class scoped_connection;
  }
}
```

Class connection

boost::signals2::connection — Query/disconnect a signal-slot connection.

Synopsis

```
// In header: <boost/signals2/connection.hpp>


class connection {
public:
  // construct/copy/destruct
  connection();
  connection(const connection&);
  connection& operator=(const connection&);

  // connection management
  void disconnect() const;
  bool connected() const;

  // blocking
  bool blocked() const;

  // modifiers
  void swap(const connection&);

  // comparisons
  bool operator==(const connection&) const;
  bool operator!=(const connection&) const;
  bool operator<(const connection&) const;
};

// specialized algorithms
void swap(connection&, connection&);
```

Description

The signals2::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.

Thread Safety

The methods of the `connection` class are thread-safe with the exception of swap and the assignment operator. A `connection` object should not be accessed concurrently when either of these operations is in progress. However, it is always safe to access a different `connection` object in another thread, even if the two `connection` objects are copies of each other which refer to the same underlying connection.

`connection` **public 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` **blocking**

1.
```
bool blocked() const;
```

Queries if the connection is blocked. A connection may be blocked by creating a `boost::signals2::shared_connection_block` object.
Returns: `true` if the associated slot is either disconnected or blocked, `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: `!(*this == other)`

Throws: Will not throw.

3.

```
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::signals2::scoped_connection — Limits a signal-slot connection lifetime to a particular scope.

Synopsis

```cpp
// In header: <boost/signals2/connection.hpp>


class scoped_connection : public connection {
public:
  // construct/copy/destruct
  scoped_connection();
  scoped_connection(const connection&);
  ~scoped_connection();

  // public methods
  scoped_connection & operator=(const connection &);
  connection release();
private:
  // construct/copy/destruct
  scoped_connection(const scoped_connection&);
  scoped_connection& operator=(const scoped_connection&);
};
```

Description

A connection which automatically disconnects on destruction.

Thread Safety

The methods of the `scoped_connection` class (including those inherited from its base `connection` class) are thread-safe with the exception of signals2::connection::swap, release, and the assignment operator. A `scoped_connection` object should not be accessed concurrently when any of these operations is in progress. However, it is always safe to access a different `connection` object in another thread, even if it references the same underlying signal-slot connection.

`scoped_connection` public construct/copy/destruct

1.
```
scoped_connection();
```

Default constructs an empty scoped_connection.

Postconditions:	connected() == false
Throws:	Will not throw.

2.
```
scoped_connection(const connection& other);
```

Effects:	this references the connection referenced by other.
Postconditions:	connected() == other.connected()
Throws:	Will not throw.

3.
```
~scoped_connection();
```

Effects: If `this->connected()`, disconnects the signal-slot connection.

`scoped_connection` public methods

1.
```
scoped_connection & operator=(const connection & rhs);
```

Effects:	this references the connection referenced by rhs. If this already references another connection, the old connection will be disconnected first.
Postconditions:	connected() == rhs.connected()

2.
```
connection release();
```

Effects:	Releases the connection so it will not be disconnected by the scoped_connection when it is destroyed or reassigned. The scoped_connection is reset to the NULL connection after this call completes.
Postconditions:	connected() == false
Returns:	A connection object referencing the connection which was released by the scoped_connection.

`scoped_connection` private construct/copy/destruct

1.
```
scoped_connection(const scoped_connection& other);
```

The scoped_connection class is not copyable. It may only be constructed from a `connection` object.

2.
```
scoped_connection& operator=(const scoped_connection& rhs);
```

The scoped_connection class is not copyable. It may only be assigned from a `connection` object.

Header <**boost/signals2/deconstruct.hpp**>

```
namespace boost {
  namespace signals2 {
    class deconstruct_access;
    class postconstructor_invoker;
    template<typename T> postconstructor_invoker<T> deconstruct();
    template<typename T, typename A1>
      postconstructor_invoker<T> deconstruct(const A1 &);
    template<typename T, typename A1, typename A2>
      postconstructor_invoker<T> deconstruct(const A1 &, const A2 &);
    template<typename T, typename A1, typename A2, ..., typename AN>
      postconstructor_invoker<T>
      deconstruct(const A1 &, const A2 &, ..., const AN &);
  }
}
```

Function deconstruct

boost::signals2::deconstruct — Create a `shared_ptr` with support for post-constructors and pre-destructors.

Synopsis

```
// In header: <boost/signals2/deconstruct.hpp>


template<typename T> postconstructor_invoker<T> deconstruct();
template<typename T, typename A1>
  postconstructor_invoker<T> deconstruct(const A1 & arg1);
template<typename T, typename A1, typename A2>
  postconstructor_invoker<T> deconstruct(const A1 & arg1, const A2 & arg2);
template<typename T, typename A1, typename A2, ..., typename AN>
  postconstructor_invoker<T>
  deconstruct(const A1 & arg1, const A2 & arg2, ..., const AN & argN);
```

Description

Creates an object and its owning `shared_ptr<T>` (wrapped inside a postconstructor_invoker) using only a single allocation, in a manner similar to that of `boost::make_shared()`. In addition, `deconstruct` supports postconstructors and predestructors. The returned `shared_ptr` is wrapped inside a postconstructor_invoker in order to provide the user with an opportunity to pass arguments to a postconstructor, while insuring the postconstructor is run before the wrapped `shared_ptr` is accessible.

In order to use `deconstruct` you must define a postconstructor for your class. More specifically, you must define an `adl_postconstruct` function which can be found via argument-dependent lookup. Typically, this means defining an `adl_postconstruct` function in the same namespace as its associated class. See the reference for postconstructor_invoker for a specification of what arguments are passed to the `adl_postconstruct` call.

Optionally, you may define a predestructor for your class. This is done by defining an `adl_predestruct` function which may be found by argument-dependent lookup. The deleter of the `shared_ptr` created by `deconstruct` will make an unqualified call to `adl_predestruct` with a single argument: a pointer to the object which is about to be deleted. As a convenience, the pointer will always be cast to point to a non-const type before being passed to `adl_predestruct`. If no user-defined `adl_predestruct` function is found via argument-dependent lookup, a default function (which does nothing) will be used. After `adl_predestruct` is called, the deleter will delete the object with `checked_delete`.

Any arguments passed to a `deconstruct()` call are forwarded to the matching constructor of the template type `T`. Arguments may also be passed to the class' associated `adl_postconstruct` function by using the postconstructor_invoker::postconstruct() methods.

Notes: If your compiler supports the C++0x features of rvalue references and variadic templates, then `deconstruct` will perform perfect forwarding of arguments to the `T` constructor, using a prototype of:

```
template< typename T, typename... Args > postconstructor_invoker< T > decon↵
struct( Args && ... args );
```

Otherwise, argument forwarding is performed via const references, as specified in the synopsis. In order to pass non-const references to a constructor, you will need to wrap them in reference wrappers using boost::ref.

You may give all the `deconstruct` overloads access to your class' private and protected constructors by declaring deconstruct_access a friend. Using private constructors in conjunction with deconstruct_access can be useful to ensure your objects are only created by `deconstruct`, and thus their postconstructors or predestructors will always be called.

Returns: A `postconstructor_invoker<T>` owning a newly allocated object of type `T`.

Class deconstruct_access

boost::signals2::deconstruct_access — Gives deconstruct access to private/protected constructors.

Synopsis

```
// In header: <boost/signals2/deconstruct.hpp>


class deconstruct_access {
};
```

Description

Declaring `deconstruct_access` a friend to your class will give the deconstruct factory function access to your class' private and protected constructors. Using private constructors in conjunction with `deconstruct_access` can be useful to ensure postconstructible or predestructible objects are always created properly using `deconstruct`.

Class postconstructor_invoker

boost::signals2::postconstructor_invoker — Pass arguments to and run postconstructors for objects created with deconstruct().

Synopsis

```
// In header: <boost/signals2/deconstruct.hpp>


class postconstructor_invoker {
public:

  // public methods
  operator const shared_ptr<T> &();
  const shared_ptr<T> & postconstruct();
  template<typename A1> const shared_ptr<T> & postconstruct(A1);
  template<typename A1, typename A2>
    const shared_ptr<T> & postconstruct(A1, A1);
  template<typename A1, typename A2, ..., typename AN>
    const shared_ptr<T> & postconstruct(A1, A1, ..., A1);
};
```

Description

Objects of type postconstructor_invoker are returned by calls to the deconstruct() factory function. These objects are intended to either be immediately assigned to a shared_ptr (in which case the class' conversion operator will perform the conversion by calling the postconstruct with no arguments), or to be converted to shared_ptr explicitly by the user calling one of the postconstruct methods.

postconstructor_invoker public methods

1.
```
operator const shared_ptr<T> &();
```

The conversion operator has the same effect as explicitly calling the postconstruct method with no arguments.

2.
```
const shared_ptr<T> & postconstruct();
template<typename A1> const shared_ptr<T> & postconstruct(A1 a1);
template<typename A1, typename A2>
  const shared_ptr<T> & postconstruct(A1 a1, A1 a2);
template<typename A1, typename A2, ..., typename AN>
  const shared_ptr<T> & postconstruct(A1 a1, A1 a2, ..., A1 aN);
```

The postconstruct methods make an unqualified call to adl_postconstruct() and then return the shared_ptr which was wrapped inside the postconstructor_invoker object by deconstruct(). The first two arguments passed to the adl_postconstruct() call are always the shared_ptr owning the object created by deconstruct(), followed by a ordinary pointer to the same object. As a convenience, the ordinary pointer will always be cast to point to a non-const type before being passed to adl_postconstruct. The remaining arguments passed to adl_postconstruct are whatever arguments the user may have passed to the postconstruct method.

Header <boost/signals2/dummy_mutex.hpp>

```
namespace boost {
  namespace signals2 {
    class dummy_mutex;
  }
}
```

Class dummy_mutex

boost::signals2::dummy_mutex — Fake mutex which does nothing.

Synopsis

```
// In header: <boost/signals2/dummy_mutex.hpp>


class dummy_mutex : public noncopyable {
public:
  void lock();
  bool try_lock();
  void unlock();
};
```

Description

You may wish to use the dummy_mutex class for the Mutex template type of your signals if you are not concerned about thread safety. This may give slightly faster performance, since dummy_mutex performs no actual locking.

```
void lock();
```

No effect.

```
bool try_lock();
```

No effect.

Returns: true.

```
void unlock();
```

No effect.

Header <boost/signals2/last_value.hpp>

```
namespace boost {
  namespace signals2 {
    template<typename T> class last_value;

    template<> class last_value<void>;

    class no_slots_error;
  }
}
```

Class template last_value

boost::signals2::last_value — Evaluate an InputIterator sequence and return the last value in the sequence.

Synopsis

```
// In header: <boost/signals2/last_value.hpp>

template<typename T>
class last_value {
public:
  // types
  typedef T result_type;

  // invocation
  template<typename InputIterator>
    result_type operator()(InputIterator, InputIterator) const;
};
```

Description

The last_value class was the default Combiner template parameter type for signals in the original Signals library. Signals2 uses optional_last_value as the default, which does not throw.

last_value invocation

1.
```
template<typename InputIterator>
  result_type operator()(InputIterator first, InputIterator last) const;
```

Effects:	Attempts to dereference every iterator in the sequence `[first, last)`.
Returns:	The result of the last successful iterator dereference.
Throws:	no_slots_error if no iterators were successfully dereferenced, unless the template type of `last_value` is `void`.

Specializations

• Class last_value<void>

Class last_value<void>

boost::signals2::last_value<void> — Evaluate an InputIterator sequence.

Synopsis

```
// In header: <boost/signals2/last_value.hpp>


class last_value<void> {
public:
  // types
  typedef void 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:	Attempts to dereference every iterator in the sequence `[first, last)`.
Throws:	Unlike the non-void versions of `last_value`, the void specialization does not throw.

Class no_slots_error

boost::signals2::no_slots_error — Indicates a combiner was unable to synthesize a return value.

Synopsis

```
// In header: <boost/signals2/last_value.hpp>


class no_slots_error : public std::exception {
public:
  virtual const char *  what() const;
};
```

Description
The `no_slots_error` exception may be thrown by signals2::last_value when it is run but unable to obtain any results from its input iterators.

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

Header <boost/signals2/mutex.hpp>

```
namespace boost {
  namespace signals2 {
    class mutex;
  }
}
```

Class mutex

boost::signals2::mutex — A header-only mutex which implements the Lockable concept of Boost.Thread.

Synopsis

```
// In header: <boost/signals2/mutex.hpp>


class mutex {
public:
  void lock();
  bool try_lock();
  void unlock();
};
```

Description

The `mutex` class implements the `Lockable` concept of Boost.Thread, and is the default `Mutex` template parameter type for signals. If boost has detected thread support in your compiler, the `mutex` class will map to a CRITICAL_SECTION on Windows or a pthread_mutex on POSIX. If thread support is not detected, `mutex` will behave similarly to a dummy_mutex. The header file `boost/config.hpp` defines the macro `BOOST_HAS_THREADS` when boost detects threading support. The user may globally disable thread support in boost by defining BOOST_DISABLE_THREADS before any boost header files are included.

If you are already using the Boost.Thread library, you may prefer to use its boost::mutex class instead as the mutex type for your signals.

You may wish to use a thread-unsafe signal, if the signal is only used by a single thread. In that case, you may prefer to use the signals2::dummy_mutex class as the `Mutex` template type for your signal.

```
void lock();
```

Locks the mutex.

```
bool try_lock();
```

Makes a non-blocking attempt to lock the mutex.

Returns: `true` on success.

```
void unlock();
```

Unlocks the mutex.

29

Header <boost/signals2/optional_last_value.hpp>

```
namespace boost {
  namespace signals2 {
    template<typename T> class optional_last_value;

    template<> class optional_last_value<void>;
  }
}
```

Class template optional_last_value

boost::signals2::optional_last_value — Evaluate an InputIterator sequence and return a boost::optional which contains the last value in the sequence, or an empty boost::optional if the sequence was empty.

Synopsis

```
// In header: <boost/signals2/optional_last_value.hpp>

template<typename T>
class optional_last_value {
public:
  // types
  typedef boost::optional<T> result_type;

  // invocation
  template<typename InputIterator>
    result_type operator()(InputIterator, InputIterator) const;
};
```

Description

`optional_last_value` is the default Combiner template type for signals in the Boost.Signals2 library. The advantage of `optional_last_value` over signals2::last_value is that `optional_last_value` can return an empty boost::optional. rather than throwing an exception, when its `InputIterator` sequence is empty.

optional_last_value invocation

1.
```
template<typename InputIterator>
  result_type operator()(InputIterator first, InputIterator last) const;
```

Effects: Attempts to dereference every iterator in the sequence [first, last).
Returns: The result of the last successful iterator dereference, wrapped in a boost::optional. The returned `optional` will be empty if no iterators were dereferenced.
Throws: Does not throw.

Specializations

• Class optional_last_value<void>

Class optional_last_value<void>

boost::signals2::optional_last_value<void> — Evaluate an InputIterator sequence.

Synopsis

```
// In header: <boost/signals2/optional_last_value.hpp>


class optional_last_value<void> {
public:
  // types
  typedef void result_type;

  // invocation
  template<typename InputIterator>
    result_type operator()(InputIterator, InputIterator) const;
};
```

Description

This specialization of signals2::optional_last_value is provided to cope with the fact that there is no such thing as an optional<void>, which optional_last_value would otherwise try to use as its result_type. This specialization instead sets the result_type to be void.

optional_last_value invocation

1.
```
template<typename InputIterator>
  result_type operator()(InputIterator first, InputIterator last) const;
```

    Effects: Attempts to dereference every iterator in the sequence [first, last).

Header <boost/signals2/shared_connection_block.hpp>

```
namespace boost {
  namespace signals2 {
    class shared_connection_block;
  }
}
```

Class shared_connection_block

boost::signals2::shared_connection_block — Blocks a connection between a signal and a slot.

Synopsis

```
// In header: <boost/signals2/shared_connection_block.hpp>


class shared_connection_block {
public:
  // construct/copy/destruct
  shared_connection_block(const boost::signals2::connection & = connection(),
                          bool = true);
  shared_connection_block(const boost::signals2::shared_connection_block &);
  shared_connection_block&
  operator=(const boost::signals2::shared_connection_block &);
  ~shared_connection_block();

  // connection blocking
  void unblock();
  void block();
  bool blocking() const;

  // miscellaneous methods
  boost::signals2::connection connection() const;
};
```

Description

A `shared_connection_block` object blocks a connection, preventing the associated slot from executing when the associated signal is invoked. The connection will remain blocked until every `shared_connection_block` that references the connection releases its block. A `shared_connection_block` releases its block when it is destroyed or its unblock method is called.

A `shared_connection_block` is safe to use even after the signals2::connection object it was constructed from has been destroyed, or the connection it references has been disconnected.

Note, blocking a connection does not guarantee the associated slot has finished execution if it is already in the process of being run when the connection block goes into effect. This is similar to the behaviour of disconnect, in that blocking a connection will not wait for the connection's associated slot to complete execution. This situation may arise in a multi-threaded environment if the connection block goes into effect concurrently with signal invocation, or in a single-threaded environment if a slot blocks its own connection.

`shared_connection_block` public construct/copy/destruct

1.
```
shared_connection_block(const boost::signals2::connection & conn = connection(),
                        bool initially_blocking = true);
```

Effects:	Creates a `shared_connection_block` which can block the connection referenced by `conn`. The `shared_connection_block` will initially block the connection if and only if the `initially_blocking` parameter is `true`. The block on the connection may be released by calling the unblock method, or destroying the `shared_connection_block` object.
	Default construction of a `shared_connection_block` results in a `shared_connection_block` which references the NULL connection. Such a `shared_connection_block` is safe to use, though not particularly useful until it is assigned another `shared_connection_block` which references a real connection.
Postconditions:	`this->blocking() == initially_blocking`

2.
```
shared_connection_block(const boost::signals2::shared_connection_block & other);
```

Effects:	Copy constructs a `shared_connection_block` which references the same connection as `other`.
Postconditions:	`this->connection() == other.connection()`

```
                         this->blocking() == other.blocking()
```

3.
```
shared_connection_block&
operator=(const boost::signals2::shared_connection_block & rhs);
```

Effects:	Makes `this` reference the same connection as `rhs`.
Postconditions:	`this->connection() == rhs.connection()`
	`this->blocking() == rhs.blocking()`
Throws:	Will not throw.

4.
```
~shared_connection_block();
```

Effects: If blocking() is true, releases the connection block.

`shared_connection_block` connection blocking

1.
```
void unblock();
```

Effects:	If blocking() is true, releases the connection block. Note, the connection may remain blocked due to other `shared_connection_block` objects.
Postconditions:	`this->blocking() == false`.

2.
```
void block();
```

Effects:	If blocking() is false, reasserts a block on the connection.
Postconditions:	`this->blocking() == true`.

3.
```
bool blocking() const;
```

Returns:	`true` if `this` is asserting a block on the connection.
Notes:	`this->blocking() == true` implies connection::blocked() == true for the connection. However, `this->blocking() == false` does not necessarily imply `connection::blocked() == false`, since the connection may be blocked by another `shared_connection_block` object.

`shared_connection_block` miscellaneous methods

1.
```
boost::signals2::connection connection() const;
```

Returns: A connection object for the connection referenced by `this`.

Header <boost/signals2/signal.hpp>

```
namespace boost {
  namespace signals2 {

    enum connect_position { at_front, at_back };

    template<typename Signature,
             typename Combiner = boost::signals2::optional_last_value<R>,
             typename Group = int, typename GroupCompare = std::less<Group>,
             typename SlotFunction = boost::function<Signature>,
             typename ExtendedSlotFunction = boost::function<R (const connec↵
tion &, T1, T2, ..., TN)>,
             typename Mutex = boost::signals2::mutex>
      class signal;
  }
}
```

Class template signal

boost::signals2::signal — Safe multicast callback.

Synopsis

```
// In header: <boost/signals2/signal.hpp>

template<typename Signature,
         typename Combiner = boost::signals2::optional_last_value<R>,
         typename Group = int, typename GroupCompare = std::less<Group>,
         typename SlotFunction = boost::function<Signature>,
       typename ExtendedSlotFunction = boost::function<R (const connection &, T1, T2, ..., TN)>,
         typename Mutex = boost::signals2::mutex>
class signal : public boost::signals2::signal_base {
public:
  // types
  typedef Signature                                                       sig↵
nature_type;
  typedef typename Combiner::result_type                                  res↵
ult_type;
  typedef Combiner                                                        com↵
biner_type;
  typedef Group                                                           ↵
 group_type;
  typedef GroupCompare                                                    ↵
 group_compare_type;
  typedef SlotFunction                                                    ↵
 slot_function_type;
  typedef typename signals2::slot<Signature, SlotFunction>                ↵
 slot_type;
  typedef ExtendedSlotFunction                                           ex↵
tended_slot_function_type;
  typedef typename signals2::slot<R (const connection &, T1, ..., TN), ExtendedSlotFunction> ex↵
tended_slot_type;
  typedef typename SlotFunction::result_type                              ↵
 slot_result_type;
  typedef unspecified                                                     ↵
 slot_call_iterator;
  typedef T1                                                              ar↵
gument_type;                      // Exists iff arity == 1
  typedef T1                                                              ↵
 first_argument_type;             // Exists iff arity == 2
  typedef T2                                                              ↵
 second_argument_type;            // Exists iff arity == 2

  // static constants
  static const int arity = N;  // The number of arguments taken by the signal.

  // member classes/structs/unions
  template<unsigned n>
  class arg {
  public:
    // types
    typedef Tn type;  // The type of the signal's (n+1)th argument
  };

  // construct/copy/destruct
  signal(const combiner_type& = combiner_type(),
         const group_compare_type& = group_compare_type());
  ~signal();

  // connection management
  connection connect(const slot_type&, connect_position = at_back);
  connection connect(const group_type&, const slot_type&,
                     connect_position = at_back);
```

```
  connection connect_extended(const extended_slot_type&,
                              connect_position = at_back);
  connection connect_extended(const group_type&, const extended_slot_type&,
                              connect_position = at_back);
  void disconnect(const group_type&);
  template<typename S> void disconnect(const S&);
  void disconnect_all_slots();
  bool empty() const;
  std::size_t num_slots() const;

  // invocation
  result_type operator()(arg<0>::type, arg<1>::type, ..., arg<N-1>::type);
  result_type operator()(arg<0>::type, arg<1>::type, ..., arg<N-1>::type) const;

  // combiner access
  combiner_type combiner() const;
  void set_combiner(const combiner_type&);
};
```

Description

See the tutorial for more information on how to use the signal class.

Template Parameters

1.
```
typename Signature
```

2.
```
typename Combiner = boost::signals2::optional_last_value<R>
```

3.
```
typename Group = int
```

4.
```
typename GroupCompare = std::less<Group>
```

5.
```
typename SlotFunction = boost::function<Signature>
```

6.
```
typename ExtendedSlotFunction = boost::function<R (const connection &, T1, T2, ..., TN)>
```

7.
```
typename Mutex = boost::signals2::mutex
```

`signal` public types

1. typedef typename signals2::slot<R (const connection &, T1, ..., TN), ExtendedSlotFunction> extended_slot_type;

   Slots of the `extended_slot_type` may be connected to the signal using the connect_extended methods. The `extended_slot_type` has an additional signals2::connection argument in its signature, which gives slot functions access to their connection to the signal invoking them.

2. typedef typename SlotFunction::result_type slot_result_type;

This is the type returned when dereferencing the input iterators passed to the signal's combiner.

3. typedef *unspecified* slot_call_iterator;

The input iterator type passed to the combiner when the signal is invoked.

`signal` public construct/copy/destruct

1.
```
signal(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.
```
~signal();
```

Effects: Disconnects all slots connected to `*this`.

`signal` connection management

1.
```
connection connect(const slot_type& slot, connect_position at = at_back);
connection connect(const group_type& group, const slot_type& slot,
                   connect_position at = at_back);
```

Effects: Connects the signal this to the incoming slot. If the slot is inactive, i.e., any of the slots's tracked objects 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 signals2::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.
```
connection connect_extended(const extended_slot_type& slot,
                            connect_position at = at_back);
connection connect_extended(const group_type& group,
                            const extended_slot_type& slot,
                            connect_position at = at_back);
```

The connect_extended methods work the same as the connect methods, except they take slots of type extended_slot_type. This is useful if a slot needs to access the connection between it and the signal invoking it, for example if it wishes to disconnect or block its own connection.

3.
```
void disconnect(const group_type& group);
template<typename S> void disconnect(const S& slot_func);
```

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 function are disconnected.

Note, the slot_func argument should not be an actual signals2::slot object (which does not even support operator==), but rather the functor you wrapped inside a signals2::slot when you initially made the connection.

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.

4.
```
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.

5.
```
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.

6.
```
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.

signal invocation

1.
```
result_type operator()(arg<0>::type a1, arg<1>::type a2, ...,
                       arg<N-1>::type aN);
result_type operator()(arg<0>::type a1, arg<1>::type a2, ...,
                       arg<N-1>::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.

signal combiner access

1.
```
combiner_type combiner() const;
```

Returns:	A copy of the stored combiner.
Throws:	Will not throw.

2.
```
void set_combiner(const combiner_type& combiner);
```

Effects: Copies a new combiner into the signal for use with future signal invocations.

Throws: Will not throw.

Class template arg

boost::signals2::signal::arg

Synopsis

```
// In header: <boost/signals2/signal.hpp>


template<unsigned n>
class arg {
public:
  // types
  typedef Tn type;  // The type of the signal's (n+1)th argument
};
```

Header <boost/signals2/signal_base.hpp>

```
namespace boost {
  namespace signals2 {
    class signal_base;
  }
}
```

Class signal_base

boost::signals2::signal_base — Base class for signals.

Synopsis

```
// In header: <boost/signals2/signal_base.hpp>


class signal_base : public noncopyable {
public:
  // construct/copy/destruct
  virtual ~signal_base();
};
```

Description

signal_base public construct/copy/destruct

1.
```
virtual ~signal_base();
```

Virtual destructor.

Header <boost/signals2/signal_type.hpp>

```
namespace boost {
  namespace signals2 {
    template<typename A0, typename A1 = boost::parameter::void_,
             typename A2 = boost::parameter::void_,
             typename A3 = boost::parameter::void_,
             typename A4 = boost::parameter::void_,
             typename A5 = boost::parameter::void_,
             typename A6 = boost::parameter::void_>
      class signal_type;
    namespace keywords {
      template<typename Signature> class signature_type;
      template<typename Combiner> class combiner_type;
      template<typename Group> class group_type;
      template<typename GroupCompare> class group_compare_type;
      template<typename SlotFunction> class slot_function_type;
      template<typename ExtendedSlotFunction> class extended_slot_function_type;
      template<typename Mutex> class mutex_type;
    }
  }
}
```

Class template signature_type

boost::signals2::keywords::signature_type — A template keyword for signal_type.

Synopsis

```
// In header: <boost/signals2/signal_type.hpp>

template<typename Signature>
class signature_type : public unspecified-type {
};
```

Description

This class is a template keyword which may be used to pass the wrapped `Signature` template type to the signal_type metafunction as a named parameter.

The code for this class is generated by a calling a macro from the Boost.Parameter library: `BOOST_PARAMETER_TEM-PLATE_KEYWORD(signature_type)`

Class template combiner_type

boost::signals2::keywords::combiner_type — A template keyword for signal_type.

Synopsis

```
// In header: <boost/signals2/signal_type.hpp>

template<typename Combiner>
class combiner_type : public unspecified-type {
};
```

Description

This class is a template keyword which may be used to pass the wrapped `Combiner` template type to the signal_type metafunction as a named parameter.

The code for this class is generated by a calling a macro from the Boost.Parameter library: `BOOST_PARAMETER_TEM-PLATE_KEYWORD(combiner_type)`

Class template group_type

boost::signals2::keywords::group_type — A template keyword for signal_type.

Synopsis

```
// In header: <boost/signals2/signal_type.hpp>

template<typename Group>
class group_type : public unspecified-type {
};
```

Description

This class is a template keyword which may be used to pass the wrapped `Group` template type to the signal_type metafunction as a named parameter.

The code for this class is generated by a calling a macro from the Boost.Parameter library: `BOOST_PARAMETER_TEM-PLATE_KEYWORD(group_type)`

Class template group_compare_type

boost::signals2::keywords::group_compare_type — A template keyword for signal_type.

Synopsis

```
// In header: <boost/signals2/signal_type.hpp>

template<typename GroupCompare>
class group_compare_type : public unspecified-type {
};
```

Description

This class is a template keyword which may be used to pass the wrapped `GroupCompare` template type to the signal_type metafunction as a named parameter.

The code for this class is generated by a calling a macro from the Boost.Parameter library: `BOOST_PARAMETER_TEM-PLATE_KEYWORD(group_compare_type)`

Class template slot_function_type

boost::signals2::keywords::slot_function_type — A template keyword for signal_type.

Synopsis

```
// In header: <boost/signals2/signal_type.hpp>

template<typename SlotFunction>
class slot_function_type : public unspecified-type {
};
```

Description

This class is a template keyword which may be used to pass the wrapped SlotFunction template type to the signal_type metafunction as a named parameter.

The code for this class is generated by a calling a macro from the Boost.Parameter library: BOOST_PARAMETER_TEM-PLATE_KEYWORD(slot_function_type)

Class template extended_slot_function_type

boost::signals2::keywords::extended_slot_function_type — A template keyword for signal_type.

Synopsis

```
// In header: <boost/signals2/signal_type.hpp>

template<typename ExtendedSlotFunction>
class extended_slot_function_type : public unspecified-type {
};
```

Description

This class is a template keyword which may be used to pass the wrapped ExtendedSlotFunction template type to the signal_type metafunction as a named parameter.

The code for this class is generated by a calling a macro from the Boost.Parameter library: BOOST_PARAMETER_TEM-PLATE_KEYWORD(extended_slot_function_type)

Class template mutex_type

boost::signals2::keywords::mutex_type — A template keyword for signal_type.

Synopsis

```
// In header: <boost/signals2/signal_type.hpp>

template<typename Mutex>
class mutex_type : public unspecified-type {
};
```

Description

This class is a template keyword which may be used to pass the wrapped Mutex template type to the signal_type metafunction as a named parameter.

The code for this class is generated by a calling a macro from the Boost.Parameter library: `BOOST_PARAMETER_TEM-PLATE_KEYWORD(mutex_type)`

Class template signal_type

boost::signals2::signal_type — Specify a the template type parameters of a boost::signals2::signal using named parameters.

Synopsis

```
// In header: <boost/signals2/signal_type.hpp>

template<typename A0, typename A1 = boost::parameter::void_,
         typename A2 = boost::parameter::void_,
         typename A3 = boost::parameter::void_,
         typename A4 = boost::parameter::void_,
         typename A5 = boost::parameter::void_,
         typename A6 = boost::parameter::void_>
class signal_type {
public:
  // types
  typedef implementation-detail                          signature_type;        ↵

  typedef implementation-detail                          combiner_type;         ↵

  typedef implementation-detail                          group_type;            ↵

  typedef implementation-detail                          group_compare_type;    ↵

  typedef implementation-detail                          slot_function_type;    ↵

  typedef implementation-detail                           extended_slot_func↵
tion_type;
  typedef implementation-detail                          mutex_type;            ↵

  typedef typename signal<signature_type, combiner_type, ..., mutex_type> type; ↵

};
```

Description

The `signal_type` metafunction employs the Boost.Parameter library to allow users to specify the template type parameters of a signals2::signal using named parameters. The resulting signal type is provided through the `signal_type::type` typedef. Named template type parameters can enhance readability of code, and provide convenience for specifying classes which have a large number of template parameters.

The template type parameters may be passed positionally, similarly to passing them to the signals2::signal class directly. Or, they may be passed as named template parameters by wrapping them in one of the template keyword classes provided in the `boost::signals2::keywords` namespace. The supported template keywords are: keywords::signature_type, keywords::combiner_type, keywords::group_type, keywords::group_compare_type, keywords::slot_function_type, keywords::extended_slot_function_type, and keywords::mutex_type.

The default types for unspecified template type parameters are the same as those for the signal class.

Named template type parameters are especially convenient when you only wish to change a few of a signal's template type parameters from their defaults, and the parameters you wish to change are near the end of the signal's template parameter list. For example, if you only wish to change the `Mutex` template type parameter of a signal, you might write:

```
namespace bs2 = boost::signals2;
using bs2::keywords;
bs2::signal_type<void (), mutex_type<bs2::dummy_mutex> >::type sig;
```

For comparison, to specify the same type using the signal class directly looks like:

```
namespace bs2 = boost::signals2;
bs2::signal
  <
    void (),
    bs2::optional_last_value<void>,
    int,
    std::less<int>,
    boost::function<void ()>,
    boost::function<void (const connection &)>,
    bs2::dummy_mutex
  > sig;
```

Header <boost/signals2/slot.hpp>

```
namespace boost {
  namespace signals2 {
    template<typename Signature,
             typename SlotFunction = boost::function<R (T1, T2, ..., TN)> >
      class slot;
  }
}
```

Class template slot

boost::signals2::slot — Pass slots as function arguments, and associate tracked objects with a slot.

Synopsis

```cpp
// In header: <boost/signals2/slot.hpp>

template<typename Signature,
         typename SlotFunction = boost::function<R (T1, T2, ..., TN)> >
class slot : public boost::signals2::slot_base {
public:
  // types
  typedef R           result_type;
  typedef T1          argument_type;         // Exists iff arity == 1
  typedef T1          first_argument_type;   // Exists iff arity == 2
  typedef T2          second_argument_type;  // Exists iff arity == 2
  typedef Signature   signature_type;
  typedef SlotFunction slot_function_type;

  // static constants
  static const int arity = N;  // The number of arguments taken by the slot.

  // member classes/structs/unions
  template<unsigned n>
  class arg {
  public:
    // types
    typedef Tn type;  // The type of the slot's (n+1)th argument
  };

  // construct/copy/destruct
  template<typename Slot> slot(const Slot &);
  template<typename OtherSignature, typename OtherSlotFunction>
    slot(const slot<OtherSignature, OtherSlotFunction> &);
  template<typename Func, typename Arg1, typename Arg2, ..., typename ArgN>
    slot(const Func &, const Arg1 &, const Arg2 &, ..., const ArgN &);

  // invocation
  result_type operator()(arg<0>::type, arg<1>::_type, ..., arg<N-1>::type);
  result_type operator()(arg<0>::type, arg<1>::_type, ..., arg<N-1>::type) const;

  // tracking
  slot & track(const weak_ptr<void> &);
  slot & track(const signals2::signal_base &);
  slot & track(const signals2::slot_base &);
  template<typename ForeignWeakPtr>
    slot & track_foreign(const ForeignWeakPtr &,
                         typename weak_ptr_traits<ForeignWeakPtr>::shared_type * = 0);
  template<typename ForeignSharedPtr>
    slot & track_foreign(const ForeignSharedPtr &,
                         typename shared_ptr_traits<ForeignSharedPtr>::weak_type * = 0);

  // slot function access
  slot_function_type & slot_function();
  const slot_function_type & slot_function() const;
};
```

Description

A slot consists of a polymorphic function wrapper (boost::function by default) plus a container of `weak_ptrs` which identify the slot's "tracked objects". If any of the tracked objects expire, the slot will automatically disable itself. That is, the slot's function call operator will throw an exception instead of forwarding the function.call to the slot's polymorphic function wrapper. Additionally, a slot will automatically lock all the tracked objects as `shared_ptr` during invocation, to prevent any of them from expiring while the polymorphic function wrapper is being run.

The slot constructor will search for signals2::signal and signals2::trackable inside incoming function objects and automatically track them. It does so by applying a visitor to the incoming functors with boost::visit_each.

Template Parameters

1.
```
typename Signature
```

2.
```
typename SlotFunction = boost::function<R (T1, T2, ..., TN)>
```

`slot` public construct/copy/destruct

1.
```
template<typename Slot> slot(const Slot & target);
```

Effects: Initializes the `SlotFunction` object in `this` with `target`, which may be any function object with which a `SlotFunction` can be constructed.

               In this special case where the template type parameter `Slot` is a compatible signals2::signal type, the signal will automatically be added to the slot's tracked object list. Otherwise, the slot's tracked object list is initially empty.

2.
```
template<typename OtherSignature, typename OtherSlotFunction>
  slot(const slot<OtherSignature, OtherSlotFunction> & other_slot);
```

Effects: Initializes `this` with a copy of `other_slot`'s `OtherSlotFunction` object and tracked object list.

3.
```
template<typename Func, typename Arg1, typename Arg2, ..., typename ArgN>
  slot(const Func & f, const Arg1 & a1, const Arg2 & a2, ..., const ArgN & aN);
```

Effects: Syntactic sugar for `bind()` when the constructor is passed more than one argument. As if: `slot(boost::bind(f, a1, a2, ..., aN))`

`slot` invocation

1.
```
result_type operator()(arg<0>::type a1, arg<1>::_type a2, ...,
                       arg<N-1>::type aN);
result_type operator()(arg<0>::type a1, arg<1>::_type a2, ...,
                       arg<N-1>::type aN) const;
```

Effects: Calls the slot's `SlotFunction` object.
Returns: The result returned by the slot's `SlotFunction` object.
Throws: Any exceptions thrown by the slot's `SlotFunction` object. boost::signals2::expired_slot if any object in the tracked object list has expired.
Notes: If you have already used lock to insure the tracked objects are valid, it is slightly more efficient to use the slot_function() method and call the slot's `SlotFunction` directly.

`slot` tracking

1.
```
slot & track(const weak_ptr<void> & tracked_object);
slot & track(const signals2::signal_base & tracked_signal);
slot & track(const signals2::slot_base & tracked_slot);
```

Effects: Adds object(s) to the slot's tracked object list. Should any of the tracked objects expire, then subsequent attempts to call the slot's `operator()` or `lock()` methods will throw an signals2::expired_slot exception.

When tracking a signal, a shared_ptr internal to the signal class is used for tracking. The signal does not need to be owned by an external `shared_ptr`.

In the case of passing another slot as the argument to `track()`, only the objects currently in the other slot's tracked object list are added to the tracked object list of `this`. The other slot object itself is not tracked.

Returns: `*this`

2.
```cpp
template<typename ForeignWeakPtr>
  slot & track_foreign(const ForeignWeakPtr & tracked_object,
                       typename weak_ptr_traits<ForeignWeakPtr>::shared_type * SFINAE = 0);
template<typename ForeignSharedPtr>
  slot & track_foreign(const ForeignSharedPtr & tracked_object,
                       typename shared_ptr_traits<ForeignSharedPtr>::weak_type * SFINAE = 0);
```

Effects: The `track_foreign()` method behaves similarly to calling the track() method with a boost::shared_ptr or boost::weak_ptr argument. However, `track_foreign` is more flexible in that it will accept `shared_ptr` or `weak_ptr` classes from outside of boost (most significantly `std::shared_ptr` or `std::weak_ptr`).

In order to use a particular `shared_ptr` class with this function, a specialization of boost::signals2::shared_ptr_traits must exist for it. Also, a specialization of boost::signals2::weak_ptr_traits must be provided for its corresponding `weak_ptr` class. The `shared_ptr_traits` specialization must include a `weak_type` member typedef which specifies the corresponding `weak_ptr` type of the `shared_ptr` class. Similarly, the `weak_ptr_traits` specialization must include a `shared_type` member typedef which specifies the corresponding `shared_ptr` type of the `weak_ptr` class. Specializations for `std::shared_ptr` and `std::weak_ptr` are already provided by the signals2 library. For other `shared_ptr` classes, you must provide the specializations.

The second argument "SFINAE" may be ignored, it is used to resolve the overload between either `shared_ptr` or `weak_ptr` objects passed in as the first argument.

Returns: `*this`

`slot` slot function access

1.
```cpp
slot_function_type & slot_function();
const slot_function_type & slot_function() const;
```

Returns: A reference to the slot's underlying SlotFunction object.

Class template arg

boost::signals2::slot::arg

Synopsis

```cpp
// In header: <boost/signals2/slot.hpp>


template<unsigned n>
class arg {
public:
  // types
  typedef Tn type;  // The type of the slot's (n+1)th argument
};
```

Header <boost/signals2/slot_base.hpp>

```
namespace boost {
  namespace signals2 {
    class slot_base;
    class expired_slot;
  }
}
```

Class slot_base

boost::signals2::slot_base — Base class for slots.

Synopsis

```
// In header: <boost/signals2/slot_base.hpp>


class slot_base {
public:
  // types
  typedef std::vector<implementation-detail> locked_container_type;

  // tracking
  bool expired() const;
  locked_container_type lock() const;
};
```

Description

`slot_base` tracking

1.
```
bool expired() const;
```

   Returns: `true` if any tracked object has expired.

2.
```
locked_container_type lock() const;
```

   Returns: A container holding `shared_ptrs` to each of the slot's tracked objects. As long as the returned container is kept
                 in scope, none of the slot's tracked objects can expire.
   Throws: expired_slot if any of the slot's tracked objects have expired.

Class expired_slot

boost::signals2::expired_slot — Indicates at least one of a slot's tracked objects has expired.

Synopsis

```
// In header: <boost/signals2/slot_base.hpp>


class expired_slot : public bad_weak_ptr {
public:
  virtual const char *  what() const;
};
```

Description

The `expired_slot` exception is thrown to indicate at least one of a slot's tracked objects has expired.

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

Header <boost/signals2/trackable.hpp>

```
namespace boost {
  namespace signals2 {
    class trackable;
  }
}
```

Class trackable

boost::signals2::trackable — Provided to ease porting for code using the boost::signals::trackable class from the original Boost.Signals library.

Synopsis

```
// In header: <boost/signals2/trackable.hpp>


class trackable {
public:
  // construct/copy/destruct
  trackable();
  trackable(const trackable&);
  trackable& operator=(const trackable&);
  ~trackable();
};
```

Description

Use of the `trackable` class is not recommended for new code. The `trackable` class is not thread-safe since `trackable` objects disconnect their associated connections in the `trackable` destructor. Since the `trackable` destructor is not run until after the destructors of any derived classes have completed, that leaves open a window where a partially destructed object can still have active connections.

The preferred method of automatic connection management with Boost.Signals2 is to manage the lifetime of tracked objects with `shared_ptrs` and to use the signals2::slot::track method to track their lifetimes.

The `trackable` class provides automatic disconnection of signals and slots when objects bound in slots (via pointer or reference) are destroyed. `trackable` class may only be used as a public base class for some other class; when used as such, that class may be

---

49

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` public 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.

Thread-Safety

Introduction

The primary motivation for Boost.Signals2 is to provide a version of the original Boost.Signals library which can be used safely in a multi-threaded environment. This is achieved primarily through two changes from the original Boost.Signals API. One is the introduction of a new automatic connection management scheme relying on `shared_ptr` and `weak_ptr`, as described in the tutorial. The second change was the introduction of a `Mutex` template type parameter to the `signal` class. This section details how the library employs these changes to provide thread-safety, and the limits of the provided thread-safety.

Signals and combiners

Each signal object default-constructs a `Mutex` object to protect its internal state. Furthermore, a `Mutex` is created each time a new slot is connected to the signal, to protect the associated signal-slot connection.

A signal's mutex is automatically locked whenever any of the signal's methods are called. The mutex is usually held until the method completes, however there is one major exception to this rule. When a signal is invoked by calling `signal::operator()`, the invocation first acquires a lock on the signal's mutex. Then it obtains a handle to the signal's slot list and combiner. Next it releases the signal's mutex, before invoking the combiner to iterate through the slot list. Thus no mutexes are held by the signal while a slot is executing. This design choice makes it impossible for user code running in a slot to deadlock against any of the mutexes used internally by the Boost.Signals2 library. It also prevents slots from accidentally causing recursive locking attempts on any of the library's internal mutexes. Therefore, if you invoke a signal concurrently from multiple threads, it is possible for the signal's combiner to be invoked concurrently and thus the slots to execute concurrently.

During a combiner invocation, the following steps are performed in order to find the next callable slot while iterating through the signal's slot list.

- The `Mutex` associated with the connection to the slot is locked.

- All the tracked `weak_ptr` associated with the slot are copied into temporary `shared_ptr` which will be kept alive until the invocation is done with the slot. If this fails due to any of the `weak_ptr` being expired, the connection is automatically disconnected. Therefore a slot will never be run if any of its tracked `weak_ptr` have expired, and none of its tracked `weak_ptr` will expire while the slot is running.

- The slot's connection is checked to see if it is blocked or disconnected, and then the connection's mutex is unlocked. If the connection was either blocked or disconnected, we start again from the beginning with the next slot in the slot list. Otherwise, we commit to executing the slot when the combiner next dereferences the slot call iterator (unless the combiner should increment the iterator without ever dereferencing it).

Note that since we unlock the connection's mutex before executing its associated slot, it is possible a slot will still be executing after it has been disconnected by a `connection::disconnect()`, if the disconnect was called concurrently with signal invocation.

You may have noticed above that during signal invocation, the invocation only obtains handles to the signal's slot list and combiner while holding the signal's mutex. Thus concurrent signal invocations may still wind up accessing the same slot list and combiner concurrently. So what happens if the slot list is modified, for example by connecting a new slot, while a signal invocation is in progress concurrently? If the slot list is already in use, the signal performs a deep copy of the slot list before modifying it. Thus the a concurrent signal invocation will continue to use the old unmodified slot list, undisturbed by modifications made to the newly created deep copy of the slot list. Future signal invocations will receive a handle to the newly created deep copy of the slot list, and the old slot list will be destroyed once it is no longer in use. Similarly, if you change a signal's combiner with `signal::set_combiner` while a signal invocation is running concurrently, the concurrent signal invocation will continue to use the old combiner undisturbed, while future signal invocations will receive a handle to the new combiner.

The fact that concurrent signal invocations use the same combiner object means you need to insure any custom combiner you write is thread-safe. So if your combiner maintains state which is modified when the combiner is invoked, you may need to protect that state with a mutex. Be aware, if you hold a mutex in your combiner while dereferencing slot call iterators, you run the risk of deadlocks and recursive locking if any of the slots cause additional mutex locking to occur. One way to avoid these perils is for your combiner

to release any locks before dereferencing a slot call iterator. The combiner classes provided by the Boost.Signals2 library are all thread-safe, since they do not maintain any state across invocations.

Suppose a user writes a slot which connects another slot to the invoking signal. Will the newly connected slot be run during the same signal invocation in which the new connection was made? The answer is no. Connecting a new slot modifies the signal's slot list, and as explained above, a signal invocation already in progress will not see any modifications made to the slot list.

Suppose a user writes a slot which disconnects another slot from the invoking signal. Will the disconnected slot be prevented from running during the same signal invocation, if it appears later in the slot list than the slot which disconnected it? This time the answer is yes. Even if the disconnected slot is still present in the signal's slot list, each slot is checked to see if it is disconnected or blocked immediately before it is executed (or not executed as the case may be), as was described in more detail above.

Connections and other classes

The methods of the `signals2::connection` class are thread-safe, with the exception of assignment and swap. This is achived via locking the mutex associated with the object's underlying signal-slot connection. Assignment and swap are not thread-safe because the mutex protects the underlying connection which a `signals2::connection` object references, not the `signals2::connection` object itself. That is, there may be many copies of a `signals2::connection` object, all of which reference the same underlying connection. There is not a mutex for each `signals2::connection` object, there is only a single mutex protecting the underlying connection they reference.

The `shared_connection_block` class obtains some thread-safety from the `Mutex` protecting the underlying connection which is blocked and unblocked. The internal reference counting which is used to keep track of how many `shared_connection_block` objects are asserting blocks on their underlying connection is also thread-safe (the implementation relies on `shared_ptr` for the reference counting). However, individual `shared_connection_block` objects should not be accessed concurrently by multiple threads. As long as two threads each have their own `shared_connection_block` object, then they may use them in safety, even if both `shared_connection_block` objects are copies and refer to the same underlying connection.

The `signals2::slot` class has no internal mutex locking built into it. It is expected that slot objects will be created then connected to a signal in a single thread. Once they have been copied into a signal's slot list, they are protected by the mutex associated with each signal-slot connection.

The `signals2::trackable` class does NOT provide thread-safe automatic connection management. In particular, it leaves open the possibility of a signal invocation calling into a partially destructed object if the trackable-derived object is destroyed in a different thread from the one invoking the signal. `signals2::trackable` is only provided as a convenience for porting single-threaded code from Boost.Signals to Boost.Signals2.

Frequently Asked Questions

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.

2.   Is Boost.Signals2 thread-safe?

    Yes, as long as the Mutex template parameter is not set to a fake mutex type like `boost::signals2::dummy_mutex`. Also, if your slots depend on objects which may be destroyed concurrently with signal invocation, you will need to use automatic connection management. That is, the objects will need to be owned by `shared_ptr` and passed to the slot's `track`() method before the slot is connected. The `signals2::trackable` scheme of automatic connection management is NOT thread-safe, and is only provided to ease porting of single-threaded code from Boost.Signals to Boost.Signals2.

    See the documentation section on thread-safety for more information.

Design Rationale

User-level Connection Management

Users need to have fine control over the connection of signals to slots and their eventual disconnection. The primary approach taken by Boost.Signals2 is to return a `signals2::connection` object that enables connected/disconnected query, manual disconnection, and an automatic disconnection on destruction mode (`signals2::scoped_connection`). In addition, two other interfaces are supported by the `signal::disconnect` overloaded method:

- **Pass slot to disconnect**: in this interface model, the disconnection of a slot connected with `sig.con-nect(typeof(sig)::slot_type(slot_func))` is performed via `sig.disconnect(slot_func)`. Internally, a linear search using slot comparison is performed and the slot, if found, is removed from the list. Unfortunately, querying connectedness ends up as a linear-time operation.

- **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 connection approach taken by Boost.Signals2, 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.

  - If tokens are not unique, two slots may have the same name and be indistinguishable. In environments where many connections will be made dynamically, name generation becomes an additional task for the user.

  This type of interface is supported in Boost.Signals2 via the slot grouping mechanism, and the overload of `signal::disconnect` which takes an argument of the signal's `Group` type.

Automatic Connection Management

Automatic connection management in Signals2 depends on the use of `boost::shared_ptr` to manage the lifetimes of tracked objects. This is differs from the original Boost.Signals library, which instead relied on derivation from the `boost::signals::track-able` class. The library would be notified of an object's destruction by the `boost::signals::trackable` destructor.

Unfortunately, the `boost::signals::trackable` scheme cannot be made thread safe due to destructor ordering. The destructor of an class derived from `boost::signals::trackable` will always be called before the destructor of the base `boost::sig-nals::trackable` class. However, for thread-safety the connection between the signal and object needs to be disconnected before the object runs its destructors. Otherwise, if an object being destroyed in one thread is connected to a signal concurrently invoking in another thread, the signal may call into a partially destroyed object.

We solve this problem by requiring that tracked objects be managed by `shared_ptr`. Slots keep a `weak_ptr` to every object the slot depends on. Connections to a slot are disconnected when any of its tracked `weak_ptrs` expire. Additionally, signals create their own temporary `shared_ptrs` to all of a slot's tracked objects prior to invoking the slot. This insures none of the tracked objects destruct in mid-invocation.

The new connection management scheme has the advantage of being non-intrusive. Objects of any type may be tracked using the `shared_ptr`/`weak_ptr` scheme. The old `boost::signals::trackable` scheme requires the tracked objects to be derived from the `trackable` base class, which is not always practical when interacting with classes from 3rd party libraries.

`optional_last_value` as the Default Combiner

The default combiner for Boost.Signals2 has changed from the `last_value` combiner used by default in the original Boost.Signals library. This is because `last_value` requires that at least 1 slot be connected to the signal when it is invoked (except for the `last_value<void>` specialization). In a multi-threaded environment where signal invocations and slot connections and disconnections may be happening concurrently, it is difficult to fulfill this requirement. When using `optional_last_value`, there is no requirement for slots to be connected when a signal is invoked, since in that case the combiner may simply return an empty `boost::optional`.

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 Signals2 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 Signals2 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	Push

<pre>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;
  }
};</pre>
	<pre>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;
	};</pre>

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 Signals2 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).

Connection Interfaces: += operator

Boost.Signals2 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.Signals2 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 `signals2::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.

Signals2 Mutex Classes

The Boost.Signals2 library provides 2 mutex classes: `boost::signals2::mutex`, and `boost::signals2::dummy_mutex`. The motivation for providing `boost::signals2::mutex` is simply that the `boost::mutex` class provided by the Boost.Thread library currently requires linking to libboost_thread. The `boost::signals2::mutex` class allows Signals2 to remain a header-only library. You may still choose to use `boost::mutex` if you wish, by specifying it as the `Mutex` template type for your signals.

The `boost::signals2::dummy_mutex` class is provided to allow performance sensitive single-threaded applications to minimize overhead by avoiding unneeded mutex locking.

Comparison with other Signal/Slot implementations

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.Signals2, and indeed the original 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. 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.Signals2).

• **Combiner/Marshaller interface**: the equivalent to Boost.Signals2 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.

.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.

Signals2 API Changes

Porting from Boost.Signals to Boost.Signals2

The changes made to the Boost.Signals2 API compared to the original Boost.Signals library are summarized below. We also provide some notes on dealing with each change while porting existing Boost.Signals code to Boost.Signals2.

- The namespace `boost::signals` has been replaced by `boost::signals2` to avoid conflict with the original Boost.Signals implementation, as well as the Qt "signals" macro. All the Boost.Signals2 classes are inside the `boost::signals2` namespace, unlike the original Boost.Signals which has some classes in the `boost` namespace in addition to its own `boost::signals` namespace.

  The Boost.Signals2 header files are contained in the `boost/signals2/` subdirectory instead of the `boost/signals` subdirectory used by the original Boost.Signals. Furthermore, all the headers except for the convenience header `boost/signals2.hpp` are inside the `boost/signals2/` subdirectory, unlike the original Boost.Signals which keeps a few headers in the parent `boost/` directory in addition to its own `boost/signals/` subdirectory.

  For example, the `signal` class is now in the `boost::signals2` namespace instead of the `boost` namespace, and it's header file is now at `boost/signals2/signal.hpp` instead of `boost/signal.hpp`.

  While porting, only trivial changes to `#include` directives and namespace qualifications should be required to deal with these changes. Furthermore, the new namespace and header locations for Boost.Signals2 allow it to coexist in the same program with the original Boost.Signals library, and porting can be performed piecemeal.

- Automatic connection management is now achieved through the use of `shared_ptr`/`weak_ptr` and `signals2::slot::track`(), as described in the tutorial. However, the old (thread-unsafe) Boost.Signals scheme of automatic connection management is still supported via the `boost::signals2::trackable` class.

  If you do not intend to make your program multi-threaded, the easiest porting path is to simply replace your uses of `boost::signals::trackable` as a base class with `boost::signals2::trackable`. Boost.Signals2 uses the same `boost::visit_each` mechanism to discover `trackable` objects as used by the original Boost.Signals library.

- Support for postconstructors (and predestructors) on objects managed by `shared_ptr` has been added with the `deconstruct` factory function. This was motivated by the importance of `shared_ptr` for the new connection tracking scheme, and the inability to obtain a `shared_ptr` to an object in its constructor. The use of `deconstruct` is described in the tutorial.

  The use of `deconstruct` is in no way required, it is only provided in the hope it may be useful. You may wish to use it if you are porting code where a class creates connections to its own member functions in its constructor, and you also wish to use the new automatic connection management scheme. You could then move the connection creation from the constructor to to the an `adl_postconstruct` function, where a reference to the owning `shared_ptr` is available for passing to `signals2::slot::track`. The `deconstruct` function would be used create objects of the class and run their associated `adl_postconstruct` function. You can enforce use of `deconstruct` by making the class' constructors private and declaring `deconstruct_access` a friend.

- The `signals2::slot` class takes a new `Signature` template parameter, is useable as a function object, and has some additional features to support the new Boost.Signals2 automatic connection management scheme.

  The changes to the slot class should generally not cause any porting difficulties, especially if you are using the `boost::signals2::trackable` compatibility class mentioned above. If you are converting your code over to use the new automatic connection management scheme, you will need to employ some of the new slot features, as described in the tutorial.

- The `optional_last_value` class has replaced `last_value` as the default combiner for signals.

  The `signals2::last_value` combiner is still provided, although its behavior is slightly changed in that it throws an exception when no slots are connected on signal invocation, instead of always requiring at least one slot to be connected (except for its void specialization which never required any slots to be connected).

If you are porting signals which have a `void` return type in their signature and they use the default combiner, there are no changes required. If you are using the default combiner with a non-void return type and care about the value returned from signal invocation, you will have to take into account that `optional_last_value` returns a `boost::optional` instead of a plain value. One simple way to deal with this is to use `boost::optional::operator*()` to access the value wrapped inside the returned `boost::optional`.

Alternatively, you could do a port by specifying the `Combiner` template parameter for your `signals2::signal` to be `signals2::last_value`.

- The `signals2::signal` class has an additional typedef `signals2::signal::extended_slot_type` and new `signals2::signal::connect_extended`() methods. These allow connection of slots which take an additional `signals2::connection` argument, giving them thread-safe access to their signal/slot connection when they are invoked. There is also a new `ExtendedSlotFunction` template parameter for specifying the underlying slot function type for the new extended slots.

  These additions should have no effect on porting unless you are also converting your program from a single threaded program into a multi-threaded one. In that case, if you have slots which need access to their `signals2::connection` to the signal invoking them (for example to block or disconnect their connection) you may wish to connect the slots with `signals2::signal::connect_extended`(). This also requires adding an additional connection argument to the slot. More information on how and why to use extended slots is available in the tutorial.

- The `signals2::signal` class has a new `Mutex` template parameter for specifying the mutex type used internally by the signal and its connections.

  The `Mutex` template parameter can be left to its default value of `boost::signals2::mutex` and should have little effect on porting. However, if you have a single-threaded program and are concerned about incuring a performance overhead from unneeded mutex locking, you may wish to use a different mutex for your signals such as `dummy_mutex`. See the tutorial for more information on the `Mutex` parameter.

- The `signal::combiner()` method, which formerly returned a reference to the signal's combiner has been replaced by `signals2::signal::combiner` (which now returns the combiner by value) and `signals2::signal::set_combiner`.

  During porting it should be straightforward to replace uses of the old reference-returning `signal::combiner()` function with the new "by-value" `signals2::signal::combiner` and `signals2::signal::set_combiner` functions. However, you will need to inspect each call of the `combiner` method in your code to determine if your program logic has been broken by the changed return type.

- Connections no longer have `block()` and `unblock()` methods. Blocking of connections is now accomplished by creating `shared_connection_block` objects, which provide RAII-style blocking.

  If you have existing Boost.Signals code that blocks, for example:

```
namespace bs = boost::signals;

bs::connection my_connection;
//...

my_connection.block();
do_something();
my_connection.unblock();
↵
```

then the version ported to Boost.Signals2 would look like:

```
namespace bs2 = boost::signals2;

bs2::connection my_connection;
//...

{
  bs2::shared_connection_block blocker(my_connection);
  do_something();
} // blocker goes out of scope here and releases its block on my_connection
↵
```

Signals2 API Development

Version 1.4x

Version 1.45 added `slot::track_foreign`(). This method allows tracking of objects owned by `shared_ptr` classes other than `boost::shared_ptr`, for example `std::shared_ptr`.

Version 1.40

Version 1.40 adds a few new features to the `shared_connection_block` class to make it more flexible:

- `shared_connection_block` is now default constructible.

- A `shared_connection_block` may now be constructed without immediately blocking its connection.

- The `shared_connection_block::connection`() query has been added, to provide access to the `shared_connection_blocks` associated connection.

Version 1.40 also introduces a variadic templates implementation of Signals2, which is used when Boost detects compiler support for variadic templates (variadic templates are a new feature of C++0x). This change is mostly transparent to the user, however it does introduce a few visible tweaks to the interface as described in the following.

The following library features are deprecated, and are only available if your compiler is NOT using variadic templates (i.e. BOOST_NO_CXX11_VARIADIC_TEMPLATES is defined by Boost.Config).

- The "portable syntax" signal and slot classes, i.e. signals2::signal0, signal1, etc.

- The arg1_type, arg2_type, etc. member typedefs in the `signals2::signal` and `signals2::slot` classes. They are replaced by the template member classes `signals2::signal::arg` and `signals2::slot::arg`.

Version 1.39

Version 1.39 is the first release of Boost to include the Signals2 library.

Testsuite

Acceptance tests

Test	Type	Description	If failing...
connection_test.cpp	run	Test functionality of `boost::signals2::connection` and `boost::signals2::scoped_connection` objects, including release() and swap().	
dead_slot_test.cpp	run	Ensure that calling `connect` with a slot that has already expired does not actually connect to the slot.	
deconstruct_test.cpp	run	Test postconstruction/predestruction functionality of `boost::signals2::deconstruct`.	
deletion_test.cpp	run	Test deletion of slots.	
ordering_test.cpp	run	Test slot group ordering.	
regression_test.cpp	run	Tests for various bugs that have occurred in the past, to make sure they are fixed and stay fixed.	
signal_test.cpp	run	Basic test of signal/slot connections and invocation using the `boost::signals2::signal` class template.	
track_test.cpp	run	Test automatic connection management of signals and slots.	