Boost.Signals

Douglas Gregor

Table of Contents

Introduction

The Boost.Signals library is an implementation of a managed signals and slots system. Signals represent callbacks with multiple targets, and are also called publishers or events in similar systems. Signals are connected to some set of slots, which are callback receivers (also called event targets or subscribers), which are called when the signal is

"emitted."

Signals and slots are managed, in that signals and slots (or, more properly, objects that occur as part of the slots) track all connections and are capable of automatically disconnecting signal/slot connections when either is destroyed. This enables the user to make signal/slot connections without expending a great effort to manage the lifetimes of those connections with regard to the lifetimes of all objects involved.

When signals are connected to multiple slots, there is a question regarding the relationship between the return values of the slots and the return value of the signals. Boost.Signals allows the user to specify the manner in which multiple return values are combined.

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 Signals library after having read only the *Beginner* sections. The *Intermediate* sections build on the *Beginner* sections with slightly more complex uses of the library. Finally, the *Advanced* sections detail very advanced uses of the Signals library, that often require a solid working knowledge of the *Beginner* and *Intermediate* topics; most users will not need to read the *Advanced* sections.

Compatibility Note

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

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

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

Hello, World! (Beginner)

The following example writes "Hello, World!" using signals and slots. First, we create a signal `sig`, a signal that takes no arguments and has a void return value. Next, we connect the `hello` function object to the signal using the

`connect` method. Finally, use the signal `sig` like a function to call the slots, which in turns invokes `HelloWorld::operator()` to print "Hello, World!".

Preferred syntax	**Portable syntax**

`struct HelloWorld`
`{`
`  void operator()() const`
`  {`
`  std::cout << "Hello, World!\n";`
`  }`
`};`
 	
`// ...`
 	
`// No arguments and a void return value`
`boost::signal<void ()> sig;`
 	
`// Connect a HelloWorld slot`
`HelloWorld hello;`
`sig.connect(hello);`
 	
`// Call all of the slots`
`sig();`
	`struct HelloWorld`
	`{`
	`  void operator()() const`
	`  {`
	`  std::cout << "Hello, World!\n";`
	`  }`
	`};`
	
	`// ...`
	
	`// No arguments and a void return value`
	`boost::signal0<void> sig;`
	
	`// Connect a HelloWorld slot`
	`HelloWorld hello;`
	`sig.connect(hello);`
	
	`// Call all of the slots`
	`sig();`

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.

Preferred syntax	**Portable syntax**
`boost::signal<void ()> sig;`	`boost::signal0<void> sig;`

Preferred syntax	Portable syntax

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

Now, if you compile and run this program, you might see something strange. It is possible that the output will look like this:

```
  , World!
Hello
```

The underlying reason is that the ordering of signals isn't guaranteed. The signal is free to call either the `Hello` slot or the `World` slot first, but every slot will be called unless something bad (e.g., an exception) occurs. Read on to learn how to control the ordering so that "Hello, World!" always prints as expected.

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. The Boost.Signals library allows slots to be placed into groups that are ordered in some way. For our Hello, World program, we want "Hello" to be printed before ", World!", so we put "Hello" into a group that must be executed before the group that ", World!" is in. To do this, we can supply an extra parameter at the beginning of the `connect` call that specifies the group. Group values are, by default, `int`s, and are ordered by the integer < relation. Here's how we construct Hello, World:

Preferred syntax	Portable syntax

`boost::signal<void ()> sig;`
`sig.connect(0, Hello());`
`sig.connect(1, World());`
`sig();`
	`boost::signal0<void> sig;`
	`sig.connect(0, Hello());`
	`sig.connect(1, World());`
	`sig();`

This program will correctly print "Hello, World!", because the `Hello` object is in group 0, which precedes group 1 where the `World` object resides.

The group parameter is, in fact, optional. We omitted it in the first Hello, World example because it was unnecessary when all of the slots are independent. So what happens if we mix calls to connect that use the group parameter and those that don't? The "unnamed" slots (i.e., those that have been connected without specifying a group name) go into a separate group that is special in that it follows all other groups. So if we add a new slot to our example like this:

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

sig.connect(GoodMorning());
```

... we will get the result we wanted:

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

The last interesting point with groups of slots is the behavior when multiple slots are connected in the same group. Within groups, calls to slots are unordered: if we connect slots `A` and `B` to the same signal with the same group name, either `A` or `B` will be called first (but both will be called). This is the same behavior we saw before with the second version of Hello, World, where the slots could be called in the wrong order, mangling the output.

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_sum(float x, float y)
{
  std::cout << "The sum is " << x+y << std::endl;
}

void print_product(float x, float y)
{
  std::cout << "The product is " << x*y << std::endl;
}

void print_difference(float x, float y)
{
  std::cout << "The difference is " << x-y << std::endl;
}

void print_quotient(float x, float y)
{
  std::cout << "The quotient is " << x/y << std::endl;
}
```

Preferred syntax	**Portable syntax**

`boost::signal<void (float, float)> sig;`
 	
`sig.connect(&print_sum);`
`sig.connect(&print_product);`
`sig.connect(&print_difference);`
`sig.connect(&print_quotient);`
 	
`sig(5, 3);`
	`boost::signal2<void, float, float> sig;`
	
	`sig.connect(&print_sum);`
	`sig.connect(&print_product);`
	`sig.connect(&print_difference);`
	`sig.connect(&print_quotient);`
	
	`sig(5, 3);`

This program will print out something like the following, although the ordering of the lines may differ:

```
The sum is 8
The difference is 2
The product is 15
The quotient is 1.66667
```

So any values that are given to `sig` when it is called like a function are passed to each of the slots. We have to declare the types of these values up front when we create the signal. The type `boost::signal<void (float, float)>` means that the signal has a `void` return value and takes two `float` values. Any slot connected to `sig` must therefore be able to take two `float` values.

Signal Return Values (Advanced)

Just as slots can receive arguments, they can also return values. These values can then be returned back to the caller of the signal through a *combiner*. The combiner is a mechanism that can take the results of calling slots (there many be no results or a hundred; we don't know until the program runs) and coalesces them into a single result to be returned to the caller. The single result is often a simple function of the results of the slot calls: the result of the last slot call, the maximum value returned by any slot, or a container of all of the results are some possibilities.

We can modify our previous arithmetic operations example slightly so that the slots all return the results of computing the product, quotient, sum, or difference. Then the signal itself can return a value based on these results to be printed:

Preferred syntax

```
float compute_product(float x, float y)
  { return x*y; }
float compute_quotient(float x, float y)
  { return x/y; }
float compute_sum(float x, float y)
  { return x+y; }
float compute_difference(float x, float y)
  { return x-y; }

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

sig.connect(&compute_product);
sig.connect(&compute_quotient);
sig.connect(&compute_sum);
sig.connect(&compute_difference);

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

Portable syntax

```
float compute_product(float x, float y)
  { return x*y; }
float compute_quotient(float x, float y)
  { return x/y; }
float compute_sum(float x, float y)
  { return x+y; }
float compute_difference(float x, float y)
  { return x-y; }

boost::signal2<float, float, float> sig;

sig.connect(&compute_product);
sig.connect(&compute_quotient);
sig.connect(&compute_sum);
sig.connect(&compute_difference);

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

This example program will output either `8`, `1.6667`, `15`, or `2`, depending on the order that the signals are called. This is because the default behavior of a signal that has a return type (`float`, the first template argument given to the `boost::signal` class template) is to call all slots and then return the result returned by the last slot called. This behavior is admittedly silly for this example, because slots have no side effects and the result is essentially randomly chosen from the slots.

A more interesting signal result would be the maximum of the values returned by any slot. To do this, we create a custom combiner that looks like this:

```
template<typename T>
struct maximum
{
  typedef T result_type;

  template<typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    // If there are no slots to call, just return the
    // default-constructed value
    if (first == last)
      return T();

    T max_value = *first++;
```

```
    while (first != last) {
      if (max_value < *first)
        max_value = *first;
      ++first;
    }

    return max_value;
  }
};
```

The `maximum` class template acts as a function object. Its result type is given by its template parameter, and this is the type it expects to be computing the maximum based on (e.g., `maximum<float>` would find the maximum `float` in a sequence of `float`s). When a `maximum` object is invoked, it is given an input iterator sequence `[first, last)` that includes the results of calling all of the slots. `maximum` uses this input iterator sequence to calculate the maximum element, and returns that maximum value.

We actually use this new function object type by installing it as a combiner for our signal. The combiner template argument follows the signal's calling signature:

Preferred syntax	**Portable syntax**
`boost::signal<float (float x, float y),` `  maximum<float> > sig;`	`boost::signal2<float, float, float,` `  maximum<float> > sig;`

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

```
sig.connect(&compute_quotient);
sig.connect(&compute_product);
sig.connect(&compute_sum);
sig.connect(&compute_difference);

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

The output of this program will be `15`, because regardless of the order in which the slots are called, the product of 5 and 3 will be larger than the quotient, sum, or difference.

In other cases we might want to return all of the values computed by the slots together, in one large data structure. This is easily done with a different combiner:

```
template<typename Container>
struct aggregate_values
{
  typedef Container result_type;

  template<typename InputIterator>
  Container operator()(InputIterator first, InputIterator last) const
  {
    return Container(first, last);
  }
};
```

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

Preferred syntax	**Portable syntax**
`boost::signal<float (float, float),`	`boost::signal2<float, float, float,`

Preferred syntax	Portable syntax

`aggregate_values<std::vector<float> > > sig;`
 	
`sig.connect(&compute_quotient);`
`sig.connect(&compute_product);`
`sig.connect(&compute_sum);`
`sig.connect(&compute_difference);`
 	
`std::vector<float> results = sig(5, 3);`
`std::copy(results.begin(), results.end(),`
`  std::ostream_iterator<float>(cout, " "));`
	`aggregate_values<std::vector<float> > > sig;`
	
	`sig.connect(&compute_quotient);`
	`sig.connect(&compute_product);`
	`sig.connect(&compute_sum);`
	`sig.connect(&compute_difference);`
	
	`std::vector<float> results = sig(5, 3);`
	`std::copy(results.begin(), results.end(),`
	`  std::ostream_iterator<float>(cout, " "));`

The output of this program will contain 15, 8, 1.6667, and 2 (but not necessarily in that order). 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 indefinately after they are connected. Often slots are only used to receive a few events and are then disconnected, and the programmer needs control to decide when a slot should no longer be connected.

The entry point for managing connections explicitly is the `boost::signals::connection` class. The `connection` class uniquely represents the connection between a particular signal and a particular slot. The `connected()` method checks if the signal and slot are still connected, and the `disconnect()` method disconnects the signal and slot if they are connected before it is called. Each call to the signal's `connect()` method returns a connection object, which can be used to determine if the connection still exists or to disconnect the signal and slot.

```
boost::signals::connection c = sig.connect(HelloWorld());
if (c.connected()) {
// c is still connected to the signal
  sig(); // Prints "Hello, World!"
```

```
}

c.disconnect(); // Disconnect the HelloWorld object
assert(!c.connected()); c isn't connected any more

sig(); // Does nothing: there are no connected slots
```

Scoped connections (Intermediate)

The `boost::signals::scoped_connection` class references a signal/slot connection that will be disconnected when the `scoped_connection` class goes out of scope. This ability is useful when a connection need only be temporary, e.g.,

```
{
  boost::signals::scoped_connection c = sig.connect(ShortLived());
  sig(); // will call ShortLived function object
}
sig(); // ShortLived function object no longer connected to sig
```

Automatic connection management (Intermediate)

Boost.Signals can automatically track the lifetime of objects involved in signal/slot connections, including automatic disconnection of slots when objects involved in the slot call are destroyed. For instance, consider a simple news delivery service, where clients connect to a news provider that then sends news to all connected clients as information arrives. The news delivery service may be constructed like this:

Preferred syntax

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

boost::signal<void (const NewsItem& latestNews)> deliverNews;
```

Portable syntax

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

boost::signal1<void, const NewsItem&> deliverNews;
```

Clients that wish to receive news updates need only connect a function object that can receive news items to the `deliverNews` signal. For instance, we may have a special message area in our application specifically for news, e.g.,:

```
struct NewsMessageArea : public MessageArea
{
public:
  // ...

  void displayNews(const NewsItem& news) const
  {
    messageText = news.text();
    update();
  }
};

// ...
NewsMessageArea newsMessageArea = new NewsMessageArea(/* ... */);
// ...
deliverNews.connect(boost::bind(&NewsMessageArea::displayNews,
```

```
                                    newsMessageArea, _1));
```

However, what if the user closes the news message area, destroying the `newsMessageArea` object that `deliverNews` knows about? Most likely, a segmentation fault will occur. However, with Boost.Signals one need only make `NewsMessageArea` *trackable*, and the slot involving `newsMessageArea` will be disconnected when `newsMessageArea` is destroyed. The `NewsMessageArea` class is made trackable by deriving publicly from the `boost::signals::trackable` class, e.g.:

```
struct NewsMessageArea : public MessageArea, public boost::signals::trackable
{
  // ...
};
```

At this time there is a significant limitation to the use of `trackable` objects in making slot connections: function objects built using Boost.Bind are understood, such that pointers or references to `trackable` objects passed to `boost::bind` will be found and tracked.

Warning: User-defined function objects and function objects from other libraries (e.g., Boost.Function or Boost.Lambda) do not implement the required interfaces for `trackable` object detection, and *will silently ignore any bound trackable objects*. Future versions of the Boost libraries will address this limitation.

When can disconnections occur? (Intermediate)

Signal/slot disconnections occur when any of these conditions occur:

- The connection is explicitly disconnected via the connection's `disconnect` method directly, or indirectly via the signal's `disconnect` method or `scoped_connection`'s destructor.
- A `trackable` object bound to the slot is destroyed.
- The signal is destroyed.

These events can occur at any time without disrupting a signal's calling sequence. If a signal/slot connection is disconnected at any time during a signal's calling sequence, the calling sequence will still continue but will not invoke the disconnected slot. Additionally, a signal may be destroyed while it is in a calling sequence, and which case it will complete its slot call sequence but may not be accessed directly.

Signals may be invoked recursively (e.g., a signal A calls a slot B that invokes signal A...). The disconnection behavior does not change in the recursive case, except that the slot calling sequence includes slot calls for all nested invocations of the signal.

Passing slots (Intermediate)

Slots in the Boost.Signals library are created from arbitrary function objects, and therefore have no fixed type. However, it is commonplace to require that slots be passed through interfaces that cannot be templates. Slots can be passed via the `slot_type` for each particular signal type and any function object compatible with the signature of the signal can be passed to a `slot_type` parameter. For instance:

Preferred syntax

```
class Button
{
  typedef boost::signal<void (int x, int y)> OnClick;

public:
  void doOnClick(const OnClick::slot_type& slot);
```

Preferred syntax

```
private:
  OnClick onClick;
};

void Button::doOnClick(const OnClick::slot_type& slot)
{
  onClick.connect(slot);
}

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

void f(Button& button)
{
  button.doOnClick(&printCoordinates);
}
```

Portable syntax

```
class Button
{
  typedef boost::signal2<void, int x, int y> OnClick;

public:
  void doOnClick(const OnClick::slot_type& slot);

private:
  OnClick onClick;
};

void Button::doOnClick(const OnClick::slot_type& slot)
{
  onClick.connect(slot);
}

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

void f(Button& button)
{
  button.doOnClick(&printCoordinates);
}
```

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.

Reference

Header <boost/signal.hpp>

```
namespace boost {
  template<typename R, typename T1, typename T2, ..., typename TN,
           typename Combiner = last_value<R>, typename Group = int,
           typename GroupCompare = std::less<Group>,
           typename SlotFunction = functionN<R, T1, T2, ..., TN> >
    class signalN;
```

```
  template<typename Signature, typename Allocator = std::allocator<void>,
           typename Combiner = last_value<R>, typename Group = int,
           typename GroupCompare = std::less<Group>,
           typename SlotFunction = functionN<Signature> >
    class signal;
}
```

Class template signalN

Set of safe multicast callback types.

```
template<typename R, typename T1, typename T2, ..., typename TN,
         typename Combiner = last_value<R>, typename Group = int,
         typename GroupCompare = std::less<Group>,
         typename SlotFunction = functionN<R, T1, T2, ..., TN> >
class signalN : public signals::trackable, private noncopyable {
public:
  // types
  typedef typename Combiner::result_type result_type;
  typedef Combiner                       combiner_type;
  typedef Group                          group_type;
  typedef GroupCompare                   group_compare_type;
  typedef SlotFunction                   slot_function_type;
  typedef slot<SlotFunction>             slot_type;
  typedef unspecified                    slot_result_type;
  typedef unspecified                    slot_call_iterator;
  typedef T1                             argument_type;         // If N == 1
  typedef T1                             first_argument_type;   // If N == 2
  typedef T2                             second_argument_type;  // If N == 2
  typedef T1                             arg1_type;
  typedef T2                             arg2_type;
     .
     .
     .
  typedef TN                             argN_type;

  // static constants
  static const int arity = N;

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

  // connection management
  signals::connection connect(const slot_type&);
  signals::connection connect(const group_type&, const slot_type&);
  void disconnect(const group_type&);
  void disconnect_all_slots();
  bool empty() const;

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

Description

The class template signalN covers several related classes signal0, signal1, signal2, etc., where the number suffix describes the number of function parameters the signal and its connected slots will take. Instead of enumerating all classes, a single pattern signalN will be described, where N represents the number of function parameters.

`signalN` construct/copy/destruct

1.
```
signalN(const combiner_type& combiner = combiner_type(),
        const group_compare_type& compare = group_compare_type());
```

   1. **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.

2. **Postconditions:** `this->empty()`

2.
```
~signalN();
```

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

`signalN` connection management

1.
```
signals::connection connect(const slot_type& slot);
signals::connection connect(const group_type& group, const slot_type& slot);
```

1. **Effects:** Connects the signal this to the incoming slot. If the slot is inactive, i.e., any of the trackable objects bound by the slot call have been destroyed, then the call to connect is a no-op. If the second version of `connect` is invoked, the slot is associated with the given group.
2. **Returns:** A `signals::connection` object that references the newly-created connection between the signal and the slot; if the slot is inactive, returns a disconnected connection.
3. **Throws:** This routine meets the strong exception guarantee, where any exception thrown will cause the slot to not be connected to the signal.
4. **Complexity:** O(lg n) where n is the number of slots known to the signal.
5. **Notes:** It is unspecified whether connecting a slot while the signal is calling will result in the slot being called immediately.

2.
```
void disconnect(const group_type& group);
```

1. **Effects:** Any slots in the given group are disconnected.
2. **Throws:** Will not throw unless a user destructor throws. If a user destructor throws, not all slots in this group may be disconnected.
3. **Complexity:** O(lg n) + k where n is the number of slots known to the signal and k is the number of slots in the group.

3.
```
void disconnect_all_slots();
```

1. **Effects:** Disconnects all slots connected to the signal.
2. **Postconditions:** `this->empty()`.
3. **Throws:** If disconnecting a slot causes an exception to be thrown, not all slots may be disconnected.
4. **Complexity:** Linear in the number of slots known to the signal.
5. **Notes:** May be called at any time within the lifetime of the signal, including during calls to the signal's slots.

4.
```
bool empty() const;
```

1. **Returns:** `true` if no slots are connected to the signal, and `false` otherwise.
2. **Throws:** Will not throw.
3. **Complexity:** Linear in the number of slots known to the signal.

4. **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.

`signalN` invocation

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

1. **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.
2. **Returns:** The result returned by the combiner.
3. **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.
4. **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`.

   Ordering between members of a given group or between ungrouped slots is unspecified.

   Calling the function call operator may invoke undefined behavior if no slots are connected to the signal, depending on the combiner used. The default combiner is well-defined for zero slots when the return type is void but is undefined when the return type is any other type (because there is no way to synthesize a return value).

Class template signal

Safe multicast callback.

```
template<typename Signature,   // Function type R (T1, T2, ..., TN)
         typename Allocator = std::allocator<void>,
         typename Combiner = last_value<R>,
         typename Group = int,
         typename GroupCompare = std::less<Group>,
         typename SlotFunction = functionN<Signature> >
class signal : public signalN<R, T1, T2, ..., TN, Combiner, Group, GroupCompare, SlotFunction> {
public:
  // construct/copy/destruct
  signal(const combiner_type& = combiner_type(),
         const group_compare_type& = group_compare_type());
};
```

Description

Class template signal is a thin wrapper around the numbered class templates signal0, signal1, etc. It accepts a function type with N arguments instead of N separate arguments, and derives from the appropriate signalN instantiation.

All functionality of this class template is in its base class signalN.

`signal` construct/copy/destruct

1.
```
    signal(const combiner_type& combiner = combiner_type(),
           const group_compare_type& compare = group_compare_type());
```

    1. **Effects:** Initializes the base class with the given combiner and comparison objects.

Header <boost/signals/slot.hpp>

```
namespace boost {
  template<typename SlotFunction> class slot;
}
```

Class template slot

Pass slots as function arguments.

```
template<typename SlotFunction>
class slot {
public:
  // construct/copy/destruct
  template<typename Slot> slot(Slot);
};
```

Description

`slot` construct/copy/destruct

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

   1. **Effects:** Invokes `visit_each` (unqualified) to discover pointers and references to `signals::trackable` objects in `target`.

      Initializes `this` to contain the incoming slot `target`, which may be any function object with which a `SlotFunction` can be constructed.

Header <boost/signals/trackable.hpp>

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

Class trackable

Enables safe use of multicast callbacks.

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

Description

The `trackable` class provides automatic disconnection of signals and slots when objects bound in slots (via pointer or reference) are destroyed. The `trackable` class may only be used as a public base class for some other class; when used as such, that class may be bound to function objects used as part of slots. The manner in which a `trackable` object tracks the set of signal-slot connections it is a part of is unspecified.

The actual use of `trackable` is contingent on the presence of appropriate visit_each overloads for any type that may contain pointers or references to trackable objects.

`trackable` construct/copy/destruct

1. ```
   trackable();
   ```

   1. **Effects:** Sets the list of connected slots to empty.
   2. **Throws:** Will not throw.

2. ```
   trackable(const trackable& other);
   ```

   1. **Effects:** Sets the list of connected slots to empty.
   2. **Throws:** Will not throw.
   3. **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);
   ```

   1. **Effects:** Sets the list of connected slots to empty.
   2. **Returns:** `*this`
   3. **Throws:** Will not throw.
   4. **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();
   ```

   1. **Effects:** Disconnects all signal/slot connections that contain a pointer or reference to this trackable object

that can be found by visit_each.

Header <boost/signals/connection.hpp>

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

Class connection

Query/disconnect a signal-slot connection.

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

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

  // modifiers
  void swap(const connection&);

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

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

Description

The connection class represents a connection between a Signal and a Slot. It is a lightweight object that has the ability to query whether the signal and slot are currently connected, and to disconnect the signal and slot. It is always safe to query or disconnect a connection.

`connection` construct/copy/destruct

1. ```
   connection();
   ```

   1. **Effects:** Sets the currently represented connection to the NULL connection.
   2. **Postconditions:** `!this->connected()`.
   3. **Throws:** Will not throw.

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

   1. **Effects:** `this` references the connection referenced by `other`.
   2. **Throws:** Will not throw.

3. ```
   connection& operator=(const connection& other);
   ```

   1. **Effects:** `this` references the connection referenced by `other`.
   2. **Throws:** Will not throw.
   ```

`connection` connection management

1.
```
void disconnect() const;
```

   1. **Effects:** If `this->connected()`, disconnects the signal and slot referenced by this; otherwise, this operation is a no-op.
   2. **Postconditions:** `!this->connected()`.

2.
```
bool connected() const;
```

   1. **Returns:** `true` if this references a non-NULL connection that is still active (connected), and `false` otherwise.
   2. **Throws:** Will not throw.

`connection` modifiers

1.
```
void swap(const connection& other);
```

   1. **Effects:** Swaps the connections referenced in `this` and `other`.
   2. **Throws:** Will not throw.

`connection` comparisons

1.
```
bool operator==(const connection& other) const;
```

   1. **Returns:** `true` if `this` and `other` reference the same connection or both reference the NULL connection, and `false` otherwise.
   2. **Throws:** Will not throw.

2.
```
bool operator<(const connection& other) const;
```

   1. **Returns:** `true` if the connection referenced by `this` precedes the connection referenced by `other` based on some unspecified ordering, and `false` otherwise.
   2. **Throws:** Will not throw.

`connection` specialized algorithms

1.
```
void swap(connection& x, connection& y);
```

1. **Effects:** `x.swap(y)`
2. **Throws:** Will not throw.

Class scoped_connection

Limits a signal-slot connection lifetime to a particular scope.

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

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

Description

`scoped_connection` construct/copy/destruct

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

   1. **Effects:** `this` references the connection referenced by `other`.
   2. **Throws:** Will not throw.

2. ```
   ~scoped_connection();
   ```

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

`scoped_connection` connection management

1. ```
   void disconnect() const;
   ```

   1. **Effects:** If `this->connected()`, disconnects the signal and slot referenced by this; otherwise, this operation is a no-op.
   2. **Postconditions:** `!this->connected()`.

2. ```
   bool connected() const;
   ```

   1. **Returns:** `true` if this references a non-NULL connection that is still active (connected), and `false` otherwise.
   2. **Throws:** Will not throw.

Header <boost/visit_each.hpp>

```
namespace boost {
  template<typename Visitor, typename T>
    void visit_each(const Visitor&, const T&, int);
}
```

Function template visit_each

Allow limited exploration of class members.

```
template<typename Visitor, typename T>
  void visit_each(const Visitor& visitor, const T& t, int );
```

Description

The visit_each mechanism allows a visitor to be applied to every subobject in a given object. It is used by the Signals library to discover signals::trackable objects within a function object, but other uses may surface if used universally (e.g., conservative garbage collection). To fit within the visit_each framework, a visit_each overload must be supplied for each object type.

1. **Effects:** `visitor(t)`, and for every subobject `x` of `t`:

   - If `x` is a reference, `visit_each(visitor, ref(x), 0)`
   - Otherwise, `visit_each(visitor, x, 0)`

2. **Notes:** The third parameter is `long` for the fallback version of visit_each and the argument supplied to this third paramter must always be 0. The third parameter is an artifact of the widespread lack of proper function template ordering, and will be removed in the future.

   Library authors will be expected to add additional overloads that specialize the T argument for their classes, so that subobjects can be visited.

   Calls to visit_each are required to be unqualified, to enable argument-dependent lookup.

Header <boost/last_value.hpp>

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

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

Class template last_value

Evaluate an InputIterator sequence and return the last value in the sequence.

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

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

Description

`last_value` invocation

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

   1. **Requires:** `first != last`
   2. **Effects:** Dereferences every iterator in the sequence `[first, last)`.
   3. **Returns:** The result of dereferencing the iterator `last-1`.

Specializations

- Class last_value<void>

Class last_value<void>

Evaluate an InputIterator sequence.

```
class last_value<void> {
public:
  // types
  typedef unspecified result_type;

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

Description

`last_value` invocation

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

   1. **Effects:** Dereferences every iterator in the sequence `[first, last)`.

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.Signals thread-safe?

   No. Using Boost.Signals in a multithreaded concept is very dangerous, and it is very likely that the results will be less than satisfying. Boost.Signals will support thread safety in the future.

3.
   How do I get Boost.Signals to work with Qt?

   When building with Qt, the Moc keywords `signals` and `slots` are defined using preprocessor macros, causing programs using Boost.Signals and Qt together to fail to compile. Although this is a problem with Qt and not Boost.Signals, a user can use the two systems together by defining the `BOOST_SIGNALS_NAMESPACE` macro to some other identifier (e.g., `signalslib`) when building and using the Boost.Signals library. Then the namespace of the Boost.Signals library will be `boost::BOOST_SIGNALS_NAMESPACE` instead of `boost::signals`. To retain the original namespace name in translation units that do not interact with Qt, you can use a namespace alias:

```
namespace boost {
  namespace signals = BOOST_SIGNALS_NAMESPACE;
}
```

Design Overview

Type Erasure

"Type erasure", where static type information is eliminated by the use of dynamically dispatched interfaces, is used extensively within the Boost.Signals library to reduce the amount of code generated by template instantiation. Each signal must manage a list of slots and their associated connections, along with a `std::map` to map from group identifiers to their associated connections. However, instantiating this map for every token type, and perhaps within each translation unit (for some popular template instantiation strategies) increase compile time overhead and space overhead.

To combat this so-called "template bloat", we use Boost.Function and Boost.Any to store unknown types and operations. Then, all of the code for handling the list of slots and the mapping from slot identifiers to connections is factored into the class `signal_base` that deals exclusively with the `any` and `function` objects, hiding the actual implementations using the well-known pimpl idiom. The actual `signalN` class templates deal only with code that will change depending on the number of arguments or which is inherently template-dependent (such as connection).

`connection` class

The `connection` class is central to the behavior of the Boost.Signals library. It is the only entity within the Boost.Signals system that has knowledge of all objects that are associated by a given connection. To be specific, the `connection` class itself is merely a thin wrapper over a `shared_ptr` to a `basic_connection` object.

`connection` objects are stored by all participants in the Signals system: each `trackable` object contains a list of `connection` objects describing all connections it is a part of; similarly, all signals contain a set of pairs that define a slot. The pairs consist of a slot function object (generally a Boost.Function object) and a `connection` object (that will disconnect on destruction). Finally, the mapping from slot groups to slots is based on the key value in a `std::multimap` (the stored data in the `std::multimap` is the slot pair).

Slot Call Iterator

The slot call iterator is conceptually a stack of iterator adaptors that modify the behavior of the underlying iterator through the list of slots. The following table describes the type and behavior of each iterator adaptor required. Note that this is only a conceptual model: the implementation collapses all these layers into a single iterator adaptor because several popular compilers failed to compile the implementation of the conceptual model.

Iterator Adaptor	**Purpose**
Slot List Iterator	An iterator through the list of slots connected to a signal. The `value_type` of this iterator will be `std::pair<any, connection>`, where the `any` contains an instance of the slot function type.
Filter Iterator Adaptor	This filtering iterator adaptor filters out slots that have been disconnected, so we never see a disconnected slot in later stages.

Iterator Adaptor	Purpose
Projection Iterator Adaptor	The projection iterator adaptor returns a reference to the first member of the pair that constitutes a connected slot (e.g., just the `boost::any` object that holds the slot function).
Transform Iterator Adaptor	This transform iterator adaptor performs an `any_cast` to extract a reference to the slot function with the appropriate slot function type.
Transform Iterator Adaptor	This transform iterator adaptor calls the function object returned by dereferencing the underlying iterator with the set of arguments given to the signal itself, and returns the result of that slot call.
Input Caching Iterator Adaptor	This iterator adaptor caches the result of dereferencing the underlying iterator. Therefore, dereferencing this iterator multiple times will only result in the underlying iterator being dereferenced once; thus, a slot can only be called once but its result can be used multiple times.
Slot Call Iterator	Iterates over calls to each slot.

`visit_each` function template

The `visit_each` function template is a mechanism for discovering objects that are stored within another object. Function template `visit_each` takes three arguments: an object to explore, a visitor function object that is invoked with each subobject, and the `int` 0.

The third parameter is merely a temporary solution to the widespread lack of proper function template partial ordering. The primary `visit_each` function template specifies this third parameter type to be `long`, whereas any user specializations must specify their third parameter to be of type `int`. Thus, even though a broken compiler cannot tell the ordering between, e.g., a match against a parameter `T` and a parameter `A<T>`, it can determine that the conversion from the integer 0 to `int` is better than the conversion to `long`. The ordering determined by this conversion thus achieves partial ordering of the function templates in a limited, but successful, way. The following example illustrates the use of this technique:

```
template<typename> class A {};
template<typename T> void foo(T, long);
template<typename T> void foo(A<T>, int);
A<T> at;
foo(at, 0);
```

In this example, we assume that our compiler can not tell that `A<T>` is a better match than `T`, and therefore assume that the function templates cannot be ordered based on that parameter. Then the conversion from 0 to `int` is better than the conversion from 0 to `long`, and the second function template is chosen.

Design Rationale

Choice of Slot Definitions

The definition of a slot differs amongst signals and slots libraries. Within Boost.Signals, a slot is defined in a very

loose manner: it can be any function object that is callable given parameters of the types specified by the signal, and whose return value is convertible to the result type expected by the signal. However, alternative definitions have associated pros and cons that were considered prior to the construction of Boost.Signals.

- **Slots derive from a specific base class**: generally a scheme such as this will require all user-defined slots to derive from some library-specified `Slot` abstract class that defines a virtual function calling the slot. Adaptors can be used to convert a definition such as this to a definition similar to that used by Boost.Signals, but the use of a large number of small adaptor classes containing virtual functions has been found to cause an unacceptable increase in the size of executables (polymorphic class types require more code than non-polymorphic types).

  This approach does have the benefit of simplicity of implementation and user interface, from an object-oriented perspective.

- **Slots constructed from a set of primitives**: in this scheme the slot can have a limited set of types (often derived from a common abstract base class) that are constructed from some library-defined set of primitives that often include conversions from free function pointers and member function pointers, and a limited set of binding capabilities. Such an approach is reasonably simple and cover most common cases, but it does not allow a large degree of flexibility in slot construction. Libraries for function object composition have become quite advanced and it is out of the scope of a signals and slots library to encorporate such enhancements. Thus Boost.Signals does not include argument binding or function object composition primitives, but instead provides a hook (via the `visit_each` mechanism) that allows existing binder/composition libraries to provide the necessary information to Signals.

Users not satisfied with the slot definition choice may opt to replace the default slot function type with an alternative that meets their specific needs.

User-level Connection Management

Users need to have fine control over the connection of signals to slots and their eventual disconnection. The approach taken by Boost.Signals is to return a `connection` object that enables connected/disconnected query, manual disconnection, and an automatic disconnection on destruction mode. Some other possible interfaces include:

- **Pass slot to disconnect**: in this interface model, the disconnection of a slot connected with `sig.connect(slot)` is performed via `sig.disconnect(slot)`. Internally, a linear search using slot comparison is performed and the slot, if found, is removed from the list. Unfortunately, querying connectedness will generally also end up as linear-time operations. This model also fails for implementation reasons when slots become more complex than simple function pointers, member function pointers and a limited set of compositions and argument binders: to match the slot given in the call to `disconnect` with an existing slot we would need to be able to compare arbitrary function objects, which is not feasible.

- **Pass a token to disconnect**: this approach identifies slots with a token that is easily comparable (e.g., a string), enabling slots to be arbitrary function objects. While this approach is essentially equivalent to the approach taken by Boost.Signals, it is possibly more error-prone for several reasons:

  - Connections and disconnections must be paired, so the problem becomes similar to the problems incurred when pairing `new` and `delete` for dynamic memory allocation. While errors of this sort would not be catastrophic for a signals and slots implementation, their detection is generally nontrivial.

  - Tokens must be unique, otherwise two slots will have the same name and will be indistinguishable. In environments where many connections will be made dynamically, name generation becomes an additional task for the user. Uniqueness of tokens also results in an additional failure mode when attempting to connect a slot using a token that has already been used.

  - More parameterization would be required, because the token type must be user-defined. Additional parame-

terization steepens the learning curver and overcomplicates a simple interface.

This type of interface is supported in Boost.Signals via the slot grouping mechanism. It augments the `connection` object-based connection management scheme.

Combiner Interface

The Combiner interface was chosen to mimic a call to an algorithm in the C++ standard library. It is felt that by viewing slot call results as merely a sequence of values accessed by input iterators, the combiner interface would be most natural to a proficient C++ programmer. Competing interface design generally required the combiners to be constructed to conform to an interface that would be customized for (and limited to) the Signals library. While these interfaces are generally enable more straighforward implementation of the signals & slots libraries, the combiners are unfortunately not reusable (either in other signals & slots libraries or within other generic algorithms), and the learning curve is steepened slightly to learn the specific combiner interface.

The Signals formulation of combiners is based on the combiner using the "pull" mode of communication, instead of the more complex "push" mechanism. With a "pull" mechanism, the combiner's state can be kept on the stack and in the program counter, because whenever new data is required (i.e., calling the next slot to retrieve its return value), there is a simple interface to retrieve that data immediately and without returning from the combiner's code. Contrast this with the "push" mechanism, where the combiner must keep all state in class members because the combiner's routines will be invoked for each signal called. Compare, for example, a combiner that returns the maximum element from calling the slots. If the maximum element ever exceeds 100, no more slots are to be called.

Pull

```
struct pull_max {
  typedef int result_type;

  template<typename InputIterator>
  result_type
  operator()(InputIterator first,
             InputIterator last)
  {
    if (first == last)
      throw std::runtime_error("Empty!");

    int max_value = *first++;
    while(first != last && *first <= 100) {
      if (*first > max_value)
        max_value = *first;
      ++first;
    }

    return max_value;
  }
};
```

Push

```
struct push_max {
  typedef int result_type;

  push_max()
    : max_value(), got_first(false) {}

  // returns false when we want to stop
  bool operator()(int result) {
    if (result > 100)
      return false;

    if (!got_first) {
      got_first = true;
      max_value = result;
      return true;
    }

    if (result > max_value)
      max_value = result;

    return true;
  }

  int get_value() const
  {
    if (!got_first)
      throw std::runtime_error("Empty!");
    return max_value;
  }

private:
  int  max_value;
  bool got_first;
};
```

Pull	Push

There are several points to note in these examples. The "pull" version is a reusable function object that is based on an input iterator sequence with an integer `value_type`, and is very straightforward in design. The "push" model, on the other hand, relies on an interface specific to the caller and is not generally reusable. It also requires extra state values to determine, for instance, if any elements have been received. Though code quality and ease-of-use is generally subjective, the "pull" model is clearly shorter and more reusable and will often be construed as easier to write and understand, even outside the context of a signals & slots library.

The cost of the "pull" combiner interface is paid in the implementation of the Signals library itself. To correctly handle slot disconnections during calls (e.g., when the dereference operator is invoked), one must construct the iterator to skip over disconnected slots. Additionally, the iterator must carry with it the set of arguments to pass to each slot (although a reference to a structure containing those arguments suffices), and must cache the result of calling the slot so that multiple dereferences don't result in multiple calls. This apparently requires a large degree of overhead, though if one considers the entire process of invoking slots one sees that the overhead is nearly equivalent to that in the "push" model, but we have inverted the control structures to make iteration and dereference complex (instead of making combiner state-finding complex).

Connection Interfaces: += operator

Boost.Signals supports a connection syntax with the form `sig.connect(slot)`, but a more terse syntax `sig += slot` has been suggested (and has been used by other signals & slots implementations). There are several reasons as to why this syntax has been rejected:

- **It's unnecessary**: the connection syntax supplied by Boost.Signals is no less powerful that that supplied by the `+=` operator. The savings in typing (`connect()` vs. `+=`) is essentially negligible. Furthermore, one could argue that calling `connect()` is more readable than an overload of `+=`.
- **Ambiguous return type**: there is an ambiguity concerning the return value of the `+=` operation: should it be a reference to the signal itself, to enable `sig += slot1 += slot2`, or should it return a `connection` for the newly-created signal/slot connection?
- **Gateway to operators `-=`, `+`**: when one has added a connection operator `+=`, it seems natural to have a disconnection operator `-=`. However, this presents problems when the library allows arbitrary function objects to implicitly become slots, because slots are no longer comparable.

  The second obvious addition when one has `operator+=` would be to add a `+` operator that supports addition of multiple slots, followed by assignment to a signal. However, this would require implementing `+` such that it can accept any two function objects, which is technically infeasible.

`trackable` rationale

The `trackable` class is the primary user interface to automatic connection lifetime management, and its design affects users directly. Two issues stick out most: the odd copying behavior of `trackable`, and the limitation requiring users to derive from `trackable` to create types that can participate in automatic connection management.

`trackable` copying behavior

The copying behavior of `trackable` is essentially that `trackable` subobjects are never copied; instead, the copy operation is merely a no-op. To understand this, we look at the nature of a signal-slot connection and note that the connection is based on the entities that are being connected; when one of the entities is destroyed, the connection is

destroyed. Therefore, when a `trackable` subobject is copied, we cannot copy the connections because the connections don't refer to the target entity - they refer to the source entity. This reason is dual to the reason signals are non-copyable: the slots connected to them are connected to that particular signal, not the data contained in the signal.

Why derivation from `trackable`?

For `trackable` to work properly, there are two constraints:

- `trackable` must have storage space to keep track of all connections made to this object.
- `trackable` must be notified when the object is being destructed so that it can disconnect its connections.

Clearly, deriving from `trackable` meets these two guidelines. We have not yet found a superior solution.

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.Signals, and indeed Boost.Signals was strongly influenced by Karl Nelson and libsigc++. A cursory inspection of each library will find a similar syntax for the construction of signals and in the use of connections and automatic connection lifetime management. There are some major differences in design that separate these libraries:

- **Slot definitions**: slots in libsigc++ are created using a set of primitives defined by the library. These primitives allow binding of objects (as part of the library), explicit adaptation from the argument and return types of the signal to the argument and return types of the slot (libsigc++ is, by default, more strict about types than Boost.Signals). A discussion of this approach with a comparison against the approach taken by Boost.Signals is given in Choice of Slot Definitions.
- **Combiner/Marshaller interface**: the equivalent to Boost.Signals combiners in libsigc++ are the marshallers. Marshallers are similar to the "push" interface described in Combiner Interface, and a proper treatment of the topic is given there.

.NET delegates

Microsoft has introduced the .NET Framework and an associated set of languages and language extensions, one of which is the delgate. Delegates are similar to signals and slots, but they are more limited than most C++ signals and slots implemetations in that they:

- Require exact type matches between a delegate and what it is calling.
- Do not allow return types.
- Must call a method with `this` already bound.

Testsuite

Acceptance tests

Test	Type	Description
dead_slot_test.cpp	run	Ensure that calling connect with a slot that has already been disconnected via deletion does not actually connect to the slot.
deletion_test.cpp	run	Test deletion of slots.
ordering_test.cpp	run	Test slot group ordering.
signal_n_test.cpp	run	Basic test of signal/slot connections and invocation using the boost::signalN class templates.
signal_test.cpp	run	Basic test of signal/slot connections and invocation using the boost::signal class template.
trackable_test.cpp	run	Test automatic lifetime management using boost::trackable objects.