# Signals and Slots for Library TR2

Author: Douglas Gregor, Indiana University
Document number: N2086=06-0156
Date: September 10, 2006
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

## 1 Introduction

A callback is a mechanism by which a library can execute user-specified code to customize the operation of the library, e.g., by supplying operations that to be performed when elements are visited, to compare elements in a sorting routine, or to respond to asynchronous events such as information received over a network. There are various language facilities for callbacks in C++, including function pointers, member function pointers, and virtual functions. Many callback libraries are layered on top of these language facilities, providing better abstractions for callbacks. The function facility in Library TR1 [2, §3.7], for instance, is a generalized function pointer that can store and call any function object, function pointer, or member pointer with a "compatible" signature.

TR1's function is a one-to-one callback, meaning that a given function instance can only target and call one user function at a time. Function pointers and member function pointers have this same limitation, and for many tasks the limitation does not matter. Does a sorting routine that uses a callback to compare elements really need to call multiple user functions from that callback, to decide how to sort? Probably not. There are many application areas, however, where a one-to-many relationship is desirable. For instance, when applying the Model-View-Controller pattern, changes in the model should be broadcast through a single callback out to many different views. Herb Sutter argues for the introduction of "one-to-many" callbacks (and shows their implementation using function) in a short series of articles [4, 5]. This proposal introduces a library of one-to-many callbacks based on the Boost.Signals library [3], although recent versions of libsigc++ [1] implement a very similar interface.

The proposed signals library has three important features not found in all one-to-many callback systems:

- The targets of the callback can be arbitrary function objects, allowing maximal flexibility.
- The targets of the callback can return values, which can be combined in arbitrary ways (via the notion of *accumulators*) to determine the result of the callback itself.
- The targets of the callback can be automatically disconnected when any of the objects it depends on is destroyed. For instance, this means that when a View in the Model-View-Controller scheme is deleted, it will automatically be removed for any callbacks in the controller(s) to which it is attached.

This proposal does not provide complete wording for the proposed signals library. Instead, we provide a tutorial for the library and suggest that the Boost.Signals interface [3] be introduced into Library TR2 with the following changes. If the Library Working Group is interested, we will provide proposed wording for the signals and slots library. The changes we suggest to Boost.Signals are:

- Eliminate the "slot groups" feature of Boost.Signals, which drastically complicates the interface and implementation, and severely impacts performance.
- Introduce slot_iterators, as in libsigc++, that allows better control over where slots are inserted into a signal.
- Boost.Signals "combiners" become "accumulators."

## 2 Terminology

There are many different variations on one-to-many callbacks, with many different names, including multicast callbacks, publisher-subscriber, signals & slots, and the Observer pattern. We have (somewhat) arbitrarily adopted the signals & slots terminology, popularized by the Qt library [6]. There are two primary kinds of entities in a signals and slots system:

**Signal** A signal is the origin of a message, from which many attached *slots* will be called. This is the publisher in publisher-subscriber systems and is similar to the subject in the Observer pattern. Within the proposed library, a signal is an instance of the signal class template. For example, one might create a signal to represent the event of a button being clicked in a GUI. One can think of a signal as a container of slots.

**Slot** A slot is user code that will receive a message from one or more signals. A slot is a subscriber in publisher-subscriber systems or an observer in the Observer pattern. Within the proposed library, a slot is any "compatible" function object (in the same sense as we use for TR1's function facility). For example, a slot might be a function object that pops up a dialog box when a button is clicked.

## 3 Tutorial

This section describes the capabilities of the proposed library. 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, we use the signal sig like a function to call the slots, which in turn 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
signal<void ()> sig;

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

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

Calling a single slot from a signal isn't very interesting, because we could have done that with function. 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 will 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.

```
signal<void ()> sig;

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

sig();
```

By default, slots are called in first-in first-out (FIFO) order, so the output of this program will be as expected:

```
Hello, World!
```

## 3.1 Multiple Slot Arguments

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(double x, double 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;
}

signal<void (float, float)> sig;

sig.connect(&print_sum);
sig.connect(&print_product);
sig.connect(&print_difference);
sig.connect(&print_quotient);

sig(5, 3);
```

This program will print out the following:

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

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 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, but it may require implicit conversions along the way: print_difference(), for instance, relies on the implicit conversion from float to double when it is invoked.

## 3.2 Signal Return Values

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 accumulator. The accumulator 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:

```
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; }

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

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

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

This example program will output 2. This is because the default behavior of a signal that has a return type (float, in this case) is to call all slots and then return the result returned by the last slot called. This behavior is admittedly silly for this example, because slots have no side effects and the result is the last slot connected.

A more interesting signal result would be the maximum of the values returned by any slot. To do this, we create a custom accumulator 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 is 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 value.

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

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

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

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

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

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

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

```
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 accumulator:

```
signal<float (float, float), aggregate_values<std::vector<float> > > sig;

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

std::vector<float> results = sig(5, 3);
std::copy(results.begin(), results.end(),
    std::ostream_iterator<float>(cout, "\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 accumulator. The accumulator itself is a function object and its result_type member type becomes the return type of the signal.

The input iterators passed to the accumulator transform dereference operations into slot calls. Accumulators therefore have the option to invoke only some slots until some particular criterion is met. For example, in a distributed computing system, the accumulator 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 accumulator need only check the value returned when dereferencing the iterator, and return when the value is acceptable. The following accumulator 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;
  }
};
```

## 3.3 Connection Management

Thus far, we have only connected slots: we have not worried about when they might actually be disconnected, e.g., when a particular View in a Model-View-Controller implementation is destroyed. This section discusses how to manage signal-slot connections.

### 3.3.1 Disconnecting Slots

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 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.

```
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
```

### 3.3.2 Blocking Slots

Slots can be temporarily "blocked", meaning that they will be ignored when the signal is invoked but have not been disconnected. The block() member function temporarily blocks a slot, which can be unblocked via unblock(). Here is an example of blocking/unblocking slots:

```
connection c = sig.connect(HelloWorld());
sig(); // Prints "Hello, World!"

c.block(); // block the slot
assert(c.blocked());
```

```
sig(); // No output: the slot is blocked

c.unblock(); // unblock the slot
sig(); // Prints "Hello, World!"
```

### 3.3.3 Scoped Connections

The 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.,

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

### 3.3.4 Disconnecting Equivalent Slots

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

```
void foo();
void bar();

signal<void()> sig;

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

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

### 3.3.5 Automatic Connection Management

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:

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

signal<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, the propose library provides a trackable base class that one can use to automatically disconnect the slot from the signal when newMessageArea is destroyed. The NewsMessageArea class is made trackable by deriving publicly from the trackable class, e.g.:

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

Note that this feature requires the cooperation of the function objects that are used for slots. This cooperation is handled through the visit_trackable_subobjects() function template.

### 3.3.6 When Can Disconnections Occur?

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. As an example, the following case is well-formed: a dialog contains a "Cancel" button with an onClick signal. Attached to the onClick signal is a slot that deletes the dialog, destroying the button and its signal. So long as the slot itself does not try to access the signal, dialog, or button, this usage of signals and slots is correct.

## 3.4 Passing slots

Slots in the proposed 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:

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

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

private:
  OnClick onClick;
};
```

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

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

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

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.

## References

[1] libsigc++ callback framework for C++. `http://libsigc.sourceforge.net/`, September 2006.

[2] M. Austern. Proposed draft technical report on C++ library extensions. Technical Report PDTR 19768, n1745 05-0005, ISO/IEC, January 2005.

[3] D. Gregor. Boost.signals. `http://www.boost.org/doc/html/signals.html`, September 2006.

[4] H. Sutter. Generalized function pointers. *C/C++ Users Journal*, 21(8), August 2003. `http://www.ddj.com/dept/cpp/184403746`.

[5] H. Sutter. Generalizing observer. *C/C++ Users Journal*, 21(9), September 2003. `http://www.ddj.com/dept/cpp/184403873`.

[6] Trolltech. Signals and slots. `http://doc.trolltech.com/4.0/signalsandslots.html`, September 2006.