A Deeper Look at Signals and Slots

Scott Collins 2005.12.19

what are signals and slots?

There's a short answer and a long answer. We're going to have the most fun walking through the long answer, but for those who don't want to wait:

the short answer: what they are

> "**Signals and Slots**" are a generalized implementation of the Observer pattern.
>
> A **signal** is an observable event, or at least notification that the event happened.
>
> A **slot** is a potential observer, typically in the form a function to be called.
>
> You **connect** a signal to a slot to establish the observable-observer relationship.
>
> Something **emits** a signal when the advertised event or state change happens. That is, the `emitter' must call all the functions (slots) that have registered an interest in (been connected to) this event (signal).

Signals and slots have a many-to-many relationship. One signal may be connected to any number of slots. Any number of signals may be connected to the same slot.

Signals can carry additional information, e.g., a signal sent by a window upon closing (the 'windowClosing' signal) might also carry a reference back to the window; a signal sent by a slider as it is dragged would naturally want to deliver a the value of its current position. It's helpful to think of signals and slots as function signatures. A signal can be connected to any slot with a compatible signature. A signal becomes, effectively, the client's conceptual `name' for the underlying event.

There are many different implementations of signals and slots, and they vary greatly in their strengths, architecture, interfaces, and fundamental design choices. The terminology comes from Trolltech's implementation in Qt, which has used signals and slots since its initial public debut in 1994. The concept of signals and slots is so compelling that it has become a part of the computer science landscape; the pattern and terminology have been re-used again and again in various implementations. Signals and slots remain a key part of Qt's architecture; they have become fixtures in several other general purpose toolkits as well as a few popular libraries that exist solely to provide this machinery.

the long answer: how they came to be

The short answer, above, gave you some words to define signals and slots; but I prefer to show you how you might arrive at the idea of signals and slots, and how you really use them.

One fundamental concept in programming is the act of communication where one part of a program tells some other part to do something. Let's start with a very simplified world:

starting problem: a button to reload the page

```
// C++

class Button
```

```
  {
    public:
      void clicked();
        // something that happens: Buttons may be clicked
  };

class Page
  {
    public:
      void reload();
        // ...which I might want to do when a Button is clicked
  };
```

In other words, `Pages` know how to `reload()`, and `Buttons` are sometimes `clicked()`. If we can posit a `currentPage()`, then, perhaps clicking our `Button` should ask the `currentPage` to `reload()`:

```
// C++ --- making the connection directly

void
Button::clicked()
  {
    currentPage()->reload();
      // Buttons know exactly what to do when clicked
  }
```

Somehow, this isn't quite satisfying. The class name `Button` makes it seem like this is intended to be a re-usable button class. The fact that clicking it always and only asks the `currentPage` to `reload()` means it's not really re-usable at all, and perhaps should have been named `PageReloadButton`.

In fact, for completeness' sake, I should mention that this is one possible approach. If `Button::clicked()` were `virtual`, then we could have just such a class:

```
// C++ --- connecting to different actions by specializing

class Button
  {
    public:
      virtual void clicked() = 0;
        // Buttons have no idea what to do when clicked
  };

class PageReloadButton : public Button
  {
    public:
      virtual void clicked() { currentPage()->reload(); }
        // ...specialize Button to connect it to a specific action
  };
```

Well, `Button` is re-usable, but I'm still not satisfied.

introducing callbacks

Maybe we need to take a step back and remember how we would have done this in the days of C, and assuming we wanted re-usability. First, knowing that there are lots of operations we might want to do, we need a general way of thinking about such options, and we don't have classes as an abstraction to help us. So we'll use function pointers:

```
/* C --- connecting to different actions via function pointers */

void
reloadPage_action( void* )
    /* one possible action when a Button is clicked */
  {
    reloadPage(currentPage());
  }

void
loadPage_action( void* url )
    /* another possible action when a Button is clicked */
  {
    loadPage(currentPage(), (char*)url);
  }



struct Button
  {
      /* ...now I keep a (changeable) pointer to the function to be called */
    void (*actionFunc_)();
    void* actionFuncData_;
  };
```

```
void
buttonClicked( Button* button )
  {
      /* call the attached function, whatever it might be */
    if ( button && button->actionFunc_ )
      (*button->actionFunc_)(button->actionFuncData_);
  }
```

This is what is traditionally known as a callback. The calling function, `buttonClicked()`, doesn't know at compile-time what function must be called, rather it is provided at run-time a function pointer to call through. Our buttons are now re-usable, because we can attach any actions we like in the form of function pointer callbacks.

adding some type-safety

This isn't satisfying to a C++ or Java user, of course, because it's not typesafe (note the cast on `url`).

Why should we care about type-safety? An object's type is your plan for how that object should be used. By making that plan explicit, you give the compiler the power to tell you when you've violated that plan, that is, when you are `coloring outside the lines'. Code that's not type-safe is where you've lost that information, and the compiler can no longer keep you from coloring outside the lines. It may be more compelling to think about it as driving outside the lines. Your four-wheel-drive vehical can leave the road if you really want to, but in general, some hints are appreciated: a rumble-strip to wake you up when you start to drift, a speed warning so you notice when you're doing something unsafe, a seat-belt light to keep you honest, and radar to let you know when there may be trouble ahead. Even if you don't want these things for yourself, I'm pretty sure you want them for the teen-ager borrowing your car.

Back to the problem at hand. The C implementation of our problem isn't type-safe. In C++, this same pattern is usually handled by bundling the callback function and data together in a class; so expanding on our earlier C++:

```
// re-usable actions, C++ style (callback objects)

class AbstractAction
  {
    public:
      virtual void execute() = 0;
        // sub-classes re-implement this to actually do something
  };

class Button
  {
    // ...now I keep a (changeable) pointer to the action to be executed
    AbstractAction* action_;
  };

void
Button::clicked()
  {
      // execute the attached action, whatever it may be
    if ( action_ )
      action_->execute();
  }



class PageReloadAction : public AbstractAction
    // one possible action when a Button is clicked
  {
    public:
      virtual void execute() { currentPage()->reload(); }
  };

class PageLoadAction : public AbstractAction
    // another possible action when a Button is clicked
  {
    public:
      // ...
      virtual void execute() { currentPage()->load(url_); }

    private:
      std::string url_;
  };
```

Great! Our `Buttons` are now re-usable, as we can attach arbitrary actions. Our actions are typesafe, no casts required. This is all well and good, and serves many cases when you are constructing your

program. The main difference from the earlier specialization example is that now we're specializing the actions rather than the buttons. It's very similar to our C example, as we are attaching the action as data to the button. So we haven't made any great leaps yet, just step-by-step refinements of our approach.

many-to-many

The next question is: can we apply this pattern to more complicated situations, e.g., executing more than one action when a page load completes?

Some simple extrapolation leads us to machinery to easily call multiple actions:

```
class MultiAction : public AbstractAction
    // ...an action that is composed of zero or more other actions;
    // executing it is really executing each of the sub-actions
  {
    public:
      // ...
      virtual void execute();

    private:
      std::vector<AbstractAction*> actionList_;
      // ...or any reasonable collection machinery
  };

void
MultiAction::execute()
  {
      // call execute() on each action in actionList_
    std::for_each( actionList_.begin(),
                   actionList_.end(),
                   boost::bind(&AbstractAction::execute, _1) );
  }
```

There are plenty of alternatives, but this serves as an existence proof that the idea isn't unreasonable. Don't be distracted by the use of stuff from `std::` and `boost::`. We haven't really addressed the issue of ownership of the underlying action objects (intentionally). This makes it easy to say that one action can be re-used by several `Buttons` (or other consumer of `AbstractActions`). So now we've got a many-to-many system. Replacing `AbstractAction*` with `boost::shared_ptr<AbstractAction>` is one way to solve the ownership problem and still keep the many-to-many property.

but all those classes!

You may be starting to get the idea, though, that defining a new class for each action is going to become tedious. This has been a problem throughout all the C++ examples above, right from the start.

I'll have more to say about this in a bit; I just didn't want you to be stewing over it in the meantime.

specialize this; specialize that

When we started, we were specializing the `Buttons` to call different actions. By shifting the specialization to the actions, we made `Buttons` directly re-usable. Forcing actions into a specific hierarchy reduces their re-usability, though. Maybe we need to put all the specialization into the connections themselves rather than either the `Buttons` or actions.

function objects

This idea of wrapping a function in a class as we've been doing here (where `Actions` exist almost entirely to wrap the `execute` function) is pretty useful. It's used for a great many things in C++, and C++ users go to some trouble to make such classes really seem like functions. For instance, typically the wrapped function is named `operator()()`, and not `execute()` as in the examples above. That can make instances of the class seem very much like functions:

```
class AbstractAction
  {
    public:
      virtual void operator()() = 0;
```

```
  };

  // using an action (given AbstractAction& action)
action();
```

That notation is less compelling in our `list of pointers' case, but not unthinkable. The `for_each` changes a little:

```
  // previously
std::for_each( actionList_.begin(),
               actionList_.end(),
               boost::bind(&AbstractAction::execute, _1) );

  // now
std::for_each( actionList_.begin(),
               actionList_.end(),
               boost::bind(&AbstractAction::operator(), _1) );
```

We have a couple of choices for the code in `Button::clicked()`:

```
  // previously
action_->execute();

  // option 1: use the dereferenced pointer like a function
(*action_)();

  // option 2: call the function by its new name
action_->operator()();
```

You can see this is extra trouble. Is it worth it?

Just for the purposes of explaining signals and slots, moving to `operator()()` syntax is probably overkill. In the real world, though, the more uniform you can make some set of things, the more easily you can write code that applies to all of them. By standardizing on a particular shape (our function-objects look like functions) we've taken a step towards making them more re-usable in other contexts. This is particularly important when working with templates, and fundamental to working with Boost.Function, `bind`, and Template Metaprogramming (TMP).

This is part of the answer to the problem of building connections without a lot of work specializing either the signals or the slots. Templates provide a mechanism where the specialization is no harder than using a template with the right parameters, and function objects are one of the patterns our templates might take. With templates, even though specialization is happening, it's almost transparent to the client code.

loosely coupled

Let's take a step back and review the progress we've made.

We explored strategies to make it possible to call different functions from the same call-site. This is built-in to C++ via `virtual` functions, but possible using function pointers as well. When the function we want to call doesn't have the right signature, we can wrap it an an adapter object that does. We've shown at least one way to call any number (unknown at compile-time) of functions from the same call-site. We've wrapped this all in a metaphor of `sending signals' to an unknown number of listening `slots'.

We really haven't stepped very far off the road yet. The only real distinguishing features of our system are:

- a different metaphor: signals and slots
- hooking up zero or more callbacks (slots) to the same call-site (signal)
- the focus of connections is moved away from the providers, and more towards the consumers (it's less about `Buttons` calling the right thing, than about asking a `Button` to call you)

This system does foster much more loosely-coupled objects, though. `Buttons` don't have to know about `Pages`, or vice versa. Loose couplings such as these mean fewer dependencies; and the fewer dependencies the more re-usable a component can be.

Of course somebody has to know about both `Buttons` and `Pages`, just to make that connection. What if we could describe the connections to be made with data instead of in code? Then we'd really have a loose coupling, and all our code would be re-usable.

connection blueprints

What would it take to build the connections from a non-code description? If we ignore, for the moment, the idea of having more than one signature for signals and slots (e.g., right now we just have `void (*signature)()`), it's not too hard. Given hash tables to map signal names to matching connection functions and slot names to function pointers, then anyone with a pair of strings can make a connection.

There was a lot of `hand waving' in that answer, though. We really want more than one signature. I mentioned in the short answer that signals can carry extra information. That requires a signature with parameters. We haven't dealt with the difference between member and non-member functions, an invisible signature difference. We didn't pick whether we were connecting directly to slot functions or to wrapper objects; and if to wrappers, did they already exist, or did we create them on the spot? Though the underlying idea is simple, implementing such a scheme is actually pretty challenging. It's akin to instantiating objects by classname, and depending on your approach, may even require that capability. Getting signals and slots into the hash tables requires registration machinery. Once you had such a system in place, the `too many classes' problem is solved. You can manipulate keys and let the system instantiate wrappers where needed.

Adding this capability to a signals and slots implementation plainly will require much more work than we've gone to so far. Most implemenations are likely to give up compile-time type checking of the compatibility between signals and slots when connecting by keys. The cost of such a system is higher, but its applications go far beyond automating slot connections. As I mentioned above, such a system could allow instantiation of arbitrary classes, e.g., the `Button` as well as its connections. So this could be the tool for instantiating fully assembled and connected dialogs directly from a resource description. And since it makes functions available by name, it's a natural place to attach scriptability. If you happen to need all these features, then implementing such a system is well worth it; and your signals and slots benefit from the ability to describe connections in data.

An implementation that didn't need these other features might understandably omit this extra machinery. From that point of view, such an implementation is striving to remain `lightweight'. For a framework that did use all these features, implementing them together is the lightweight answer. This is one of the choices that distinguishes the popular implementations.

signals and slots in practice: qt and boost

Qt's Signals and Slots and Boost.Signals have very different design goals, resulting in very different implementations and very different strengths and weaknesses. It is quite reasonable to use both in the same program, and I explain how, below.

using signals and slots

Signals and slots are a great tool, but how best to use them? Compared to a direct function call, there are three costs of which we need to be aware. A signal-to-slot call:

- may take more time/space than a direct function call
- probably can't be inlined
- may not be as clear to a reader of the code

We get the most benefit when the loose coupling afforded by signals and slots is really appropriate to the situation: when the two ends of the connection really don't need any better knowledge of each other. The `Button`-to-action connection is a classic example. Simulations are fertile ground, for instance:

```
class Elevator
  {
    public:
      enum Direction  { DownDirection=-1, NoDirection=0, UpDirection=1 };
      enum State      { IdleState, LoadingState, MovingState };

      // ...

    // signals:
      void floorChanged( int newFloor );
      void stateChanged( State newState );
      void directionChanged( Direction newDirection );
  };
```

The `Elevator` need not know how many displays are watching it, or anything about the displays, for that matter. Every floor might have a screen, or set of lights, or needle on a dial showing the `Elevator`'s current location and direction, and some remote control panel far away might have the same information. The `Elevator` doesn't care. As it passes (or stops at) each floor, it just reports that fact by emitting the signal `floorChanged(int)`. Traffic signals in a traffic simulation are probably an even better example.

You could write your application such that every call was as signal, but that would produce something incomprehensible and inefficient. As with any tool, you'll want to strike a reasonable balance.

the qt way

There is no better description of Qt's Signals and Slots than that in Qt's documentation; however, here's a sample based on our original problem:

```
// Qt Signals and Slots

class Button : public QObject
  {
    Q_OBJECT

    Q_SIGNALS:
      void clicked();
  };

class Page : public QObject
  {
    Q_OBJECT

    public Q_SLOTS:
      void reload();
  };

  // given pointers to an actual Button and Page:
connect(button, SIGNAL(clicked()), page, SLOT(reload()));
```

the boost.signals way

There is no better description of Boost.Signals than that in Boost's documentation; however, here's a sample based on our original problem:

```
// Boost.Signals

class Button
  {
    public:
      boost::signal< void() > clicked;
  };

class Page
  {
    public:
      void reload();
  };

  // given pointers to an actual Button and Page:
button->clicked.connect( boost::bind(&Page::reload, page) );
```

a comparison

Perhaps the most important thing to notice in the Qt and Boost examples above is that neither require any classes other than `Button` and `Page`. Both systems have solved the `too many classes' problem. Let's refine and expand on our earlier analysis. We now have:

- a different metaphor with its own vocabulary: signals and slots

- hooking up zero or more callbacks (slots) to the same call-site (signal), and vice versa (many-to-many)
- the focus is on the connections, not the providers or consumers
- we don't have to invent a new class by hand just to allow a particular connection
- ...and yet the connections are still typesafe.

These five points are the core of the signals and slots concept, a core shared by both Qt and Boost. Here's a table of some key differences. I've highlighted the behaviors that I think are clear winners.

Boost.Signals	Qt Signals and Slots
a signal is an object	a signal is a named member function signature
a signal is emitted by calling it like a function	a signal is emitted by calling it like a function, you can optionally decorate it with the `emit` keyword
signals can be global, local, or member objects	signals must be members
any code with sufficient access to connect to the signal can also cause it to be emitted	only the containing object can emit the signal
a slot is any callable function or function-object	a slot is a specially designated member function
can return values collected from multiple slots	no return
synchronous	synchronous or queued
not thread-safe	thread-safe, can cross threads
auto-disconnect on slot destruction if and only if the slot is trackable	auto-disconnect on slot destruction (because all slots are trackable)
type-safe (compile-time checked)	type-safe (run-time checked)
argument-list much match exactly	slot can ignore extra arguments in signal
signals, slots may be templates	signals, slots are never templates
implemented via straight C++	implemented via (straight C++) meta-objects generated by moc
no introspection	discoverable through introspection
	invokable through meta-objects
	connections can be established automatically from resource descriptions

Most importantly, Qt's signals and slots are deeply integrated throughout the framework. They can be created, managed, and edited with Qt Designer, a GUI design tool that is, itself, a graphical drag-and-drop interface. They can be instantiated automatically, between specially named objects, even when loading UI resources dynamically.

using them together

Previously, the real obstacle to using Qt and Boost.Signals together was Qt's use of the preprocessor to define `keywords' like `signals:`, `slot:`, and `emit`. Qt has alternatives to these (the `emit` is just decoration anyway) that are more in line with the world's expectations for `#defines`, that is, they are all upper case. As of Qt 4.1, we've unified these under the `no_keywords` option, allowing you to write standard C++ and better co-mingle with other C++ libraries. You turn off the problematic lower-case symbols by adding `no_keywords` to your configuration. Here's the .pro file from my demo program.

```
# TestSignals.pro (platform independent project file, input to qmake)
#   showing how to mix Qt Signals and Slots with Boost.Signals

  #
  # Things you'll have in your .pro when you try this...
```

```
  #

CONFIG += no_keywords
  # so Qt won't #define any non-all-caps `keywords'

INCLUDEPATH += . /usr/local/include/boost-1_33_1/
  # so we can #include <boost/someheader.hpp>

macx:LIBS   += /usr/local/lib/libboost_signals-1_33_1.a
  # ...and we need to link with the Boost.Signals library.
  # This is where it lives on my Mac,
  #   other platforms would have to add a line here



  #
  # Things specific to my demo
  #

CONFIG -= app_bundle
  # so I'll build a command-line tool instead of a Mac OS X app bundle

HEADERS += Sender.h Receiver.h
SOURCES += Receiver.cpp main.cpp
```

Since you have Qt keywords turned off, you'll use Qt's upper-case macros for designating Qt signals and slots, as I do here:

```
// Sender.h

#include <QObject>
#include <string>
#include <boost/signal.hpp>

class Sender : public QObject
  {
    Q_OBJECT

    Q_SIGNALS:  // a Qt signal
      void qtSignal( const std::string& );
        // connect with
        //  QObject::connect(sender, SIGNAL(qtSignal(const std::string&)), ...

    public:     // a Boost signal for the same signature
      boost::signal< void ( const std::string& ) >  boostSignal;
        // connect with
        //  sender->boostSignal.connect(...



    public:     // an interface to make Sender emit its signals

      void
      sendBoostSignal( const std::string& message )
        {
          boostSignal(message);
        }

      void
      sendQtSignal( const std::string& message )
        {
          qtSignal(message);
        }
  };
```

I've got a `Sender`, let's make a matching `Receiver`:

```
// Receiver.h

#include <QObject>
#include <string>

class Receiver : public QObject
  {
    Q_OBJECT

    public Q_SLOTS:
      void qtSlot( const std::string& message );
        // a Qt slot is a specially marked member function
        // a Boost slot is any callable signature
  };


// Receiver.cpp

#include "Receiver.h"
#include <iostream>

void
Receiver::qtSlot( const std::string& message )
  {
    std::cout << message << std::endl;
  }
```

Now we can hook them up and test them out:

```
// main.cpp

#include <boost/bind.hpp>

#include "Sender.h"
#include "Receiver.h"

int
main( int /*argc*/, char* /*argv*/[] )
  {
    Sender* sender = new Sender;
    Receiver* receiver = new Receiver;


      // connect the boost style signal
    sender->boostSignal.connect(boost::bind(&Receiver::qtSlot, receiver, _1));

      // connect the qt style signal
    QObject::connect(sender, SIGNAL(qtSignal(const std::string&)),
                     receiver, SLOT(qtSlot(const std::string&)));


    sender->sendBoostSignal("Boost says 'Hello, World!'");
    sender->sendQtSignal("Qt says 'Hello, World!'");

    return 0;
  }
```

On my machine, I see this:

```
[506]TestSignals$ ./TestSignals
Boost says 'Hello, World!'
Qt says 'Hello, World!'
```

One difference between the two kinds of signals in this case is that since the Boost signal is visible, anyone can make it `emit'. Adding the following in `main` would be legal and work:

```
sender->boostSignal("Boost says 'Hello, World!', directly");
```

The equivalent isn't possible, directly at least, with the Qt signal, only `Sender` is allowed to emit it. I can see where you might prefer either case. The sample already shows how to get the Boost behavior from the Qt signal, by providing a public function asking for the signal to be emitted, as our `sendQtSignal` does. To get the Qt behavior from the Boost signal we need to provide similar machinery in reverse: Hide the signal, but provide a public function allowing connection. This is only a little harder, though a lot more verbose:

```
class Sender : public QObject
  {
      // just the changes...
    private:
        // our new public connect function will be much easier to understand
        //  if we simplify some of the types
      typedef boost::signal< void ( const std::string& ) > signal_type;
      typedef signal_type::slot_type                       slot_type;

      signal_type  boostSignal;
        // our signal object is now hidden


    public:

      boost::signals::connection
      connectBoostSignal( const slot_type& slot,
                          boost::signals::connect_position pos = boost::signals::at_back )
        {
          return boostSignal.connect(slot, pos);
        }
  };
```

`connectBoostSignal` isn't a particularly pretty API, and these typedefs don't lend themselves, as is, to supporting multiple signals, but both those problems are readily addressed. With a little work, I think we could come up with a more automatic mechanism for providing this `divided access' to a Boost signal. Without actually inventing the implementation, I would probably aim for something like this:

```
// WARNING: no such thing as a connect_proxy

class Sender
  {
```

```
    public:
      connect_proxy< boost::signal< void ( const std::string& ) > >
      someSignal()
        {
          return someSignal_;
            // ...automatically wrapped in the proxy
        }

    private:
      boost::signal< void ( const std::string& ) > someSignal_;
  };

sender->someSignal().connect(someSlot);
```

summary and conclusions

Signals and slots are a refinement of the Observer Pattern, a powerful metaphor, and a great tool for assembling software from components. They've been a part of the computer science landscape for over a decade, and many mature implementations exist.

The implementation in Qt is a fundamental part of Qt's architechture, and tightly integrated with its widgets, threading, introspection, scripting, meta-object machinery, and visual GUI layout tool, Qt Designer. Qt signals are member function signatures that can only be emitted by the object of which they are members. Qt slots are specially designated member functions. Qt widgets and connections can be described in non-code resources and instantiated from such resources at runtime. Qt signals and slots are built upon the introspection facility of meta-objects in Qt, which is made possible by moc, the meta-object compiler that produces meta-object classes matching user-defined classes that specifically request this by mentioning `Q_OBJECT` in their declaration.

Boost.Signals is a statically type-safe, template-based implementation of signals and slots, where signals are instances of the template `boost::signal` and slots are any callable signature. Boost.Signals stands alone, and does not require introspection, meta-objects, or external tools; but the downside of this is that Boost.Signals does not include a facility to describe connections in non-code resources.

These two implementations both have great and complementary strengths. Using them together until now has been a challenge. In Qt 4.1 and beyond, it's easy to use both if that happens to be right for your project.

Any Qt-based project with a GUI will naturally use signals and slots. You can also benefit from signals and slots in simulations and many environments where software components can be assembled with loose couplings into a larger system. As with any metaphor or technique, moderation is the key. Choose wisely where to use signals and slots and you will be rewarded with a system that's easier to understand, more flexible, highly re-usable, and working sooner.

(comments)