Signals and Slots in C++

Sarah Thompson*

March 2002

1 Introduction

This paper introduces the `sigslot` library, which implements a type-safe, thread-safe signal/slot mechanism in C++. The library is implemented entirely in C++, and does not require source code to be preprocessed[1] in order for it to be used.

The `sigslot` library's home page is at `http://sigslot.sourceforge.net/`. Look there first for the latest version of this paper, as well as up-to-date downloads of the library itself.

1.1 The Signal/Slot Paradigm

Most traditional C++ code ultimately boils down to a (possibly large) number of classes, which interoperate by calling each other's member functions. Allowing classes to interoperate in this way conventionally requires classes to 'know about' each other in some detail. For example, a home automation system might contain a couple of classes like the following:

```
class Switch
{
public:
   virtual void Clicked() = 0;
};

class Light
{
public:
   void ToggleState();
   void TurnOn();
   void TurnOff();
};
```

If we wanted to 'wire up' a switch to a light so that clicking the switch toggled the state of a light, assuming we can't modify either Switch or Light directly, we'd need to do something like:

```
class ToggleSwitch : public Switch
{
public:
    ToggleSwitch(Light& lp)
    {
        m_lp = lp;
    }

    virtual void Clicked()
    {
        m_lp.ToggleState();
    }
```

*Email: `sarah@telergy.com` Personal Web Site: `http://www.findatlantis.com/`

[1] This requirement is the usual complaint levelled against the signal/slot mechanism in the Qt library, which requires source to be preprocessed by a proprietary tool, '`moc`'.

```
private:
    Light& m_lp;
};

Light lp1, lp2;
ToggleSwitch tsw1(lp1), tsw2(lp2);
```

which is fair enough as far as it goes, but hardly pretty.

A better solution is to use signals and slots. Signals and slots allow classes to be designed without needing to worry in too much detail how they will be connected together. Here is an alternative implementation of `Switch` and `Light`:

```
class Switch
{
public:
    signal0<> Clicked;
};

class Light : public has_slots<>
{
public:
    void ToggleState();
    void TurnOn();
    void TurnOff();
};

Switch sw1, sw2;
Light lp1, lp2;
```

The main changes here are the pure virtual function, '`Clicked()`' has gone, to be replaced by a *signal*. The `Light` class is largely unchanged, except that it inherits '`has_slots`'.

Rather than needing to implement a messy derived class like `ToggleSwitch`, it is now possible to 'wire up' the switches to the lights:

```
sw1.Clicked.connect(&lp1, &Light::ToggleState);
sw2.Clicked.connect(&lp2, &Light::ToggleState);
```

It starts becoming very clear how much power is inherent in the signal/slot approach if you later decide you want to add two extra lights, each with their own toggle switch, plus a global 'all lights on' switch and a global 'all lights off' switch:

```
Switch sw3, sw4, all_on, all_off;
Light lp3, lp4;

sw3.Clicked.connect(&lp3, &Light::ToggleState);
sw4.Clicked.connect(&lp4, &Light::ToggleState);

all_on.Clicked.connect(&lp1, &Light::TurnOn());
all_on.Clicked.connect(&lp2, &Light::TurnOn());
all_on.Clicked.connect(&lp3, &Light::TurnOn());
all_on.Clicked.connect(&lp4, &Light::TurnOn());

all_off.Clicked.connect(&lp1, &Light::TurnOff());
all_off.Clicked.connect(&lp2, &Light::TurnOff());
all_off.Clicked.connect(&lp3, &Light::TurnOff());
all_off.Clicked.connect(&lp4, &Light::TurnOff());
```

1.2 Argument Types

Signals and slots can optionally take one or more arguments, with arbitrary types. The library builds upon the C++ template mechanism, which means that signal and slot declarations, as well as calls to

signals and slots, are fully type-checked.

The naming convention used is as follows:

`signal`*n*`<`*type1*`,` *type2*`, ...>`

where *n* is the number of type arguments[2].

In the following example, a class encapsulating a window emits a variety of signals when the window is moved, resized, opened or closed:

```
class Window
{
public:
    enum WindowState { Minimised, Normal, Maximised };
    signal1<WindowState> StateChanged;
    signal2<int, int> MovedTo;
    signal2<int, int> Resized;
};

class MyControl : public Control, public has_slots<>
{
public:
    void OnStateChanged(WindowState ws);
    void OnMovedTo(int x, int y);
    void OnResize(int x, int y);
};

Window w;
MyControl c;

w.StateChanged.connect(&c, &MyControl::OnStateChanged);
w.MovedTo.connect(&c, &MyControl::OnMovedTo);
w.Resized.connect(&c, &MyControl::OnResize);
```

It is worth remembering that only the types of signals and slots need to agree in order for a connection to be made - the names used don't matter.

2 Using the Library

2.1 Emitting Signals

When a signal is fired, this is usually called *emitting* a signal. The signal declaration:

```
signal1<char *, int> ReportError;
```

can be caused to emit a signal by calling its function operator. In practice, this looks just like calling a function, although of course rather more than this happens internally:

```
ReportError("Something went wrong", ERR_SOMETHING_WRONG);
```

Alternatively, you can get an exactly equivalent result by calling the `emit()` member function:

```
ReportError.emit("Something went wrong", ERR_SOMETHING_WRONG);
```

2.2 Connecting Signals

Signals are connected by calling the signal's `connect()` member function. `connect()` takes two arguments: a pointer to the destination class, and a pointer to a member function of the destination class. In order for this to work, it is necessary for all types to agree, and also for the destination class to inherit '`has_slots`'.

[2] It would definitely be nicer if C++ would allow us to just call all of the `signal` variants '`signal`'. There doesn't appear to be any efficient way to do this within the constraints of the current ISO C++ standard, unfortunately.

A signal can be connected to any number of slots. When a signal's `emit` function is called, *all* connected slots get called. This is why slots always have a `void` return type – since zero or many slots are called when a signal emits, it doesn't make sense to implement a return value[3].

The current implementation uses an STL list to implement the list of connected slots. This means that slots are called in the same order that they were connected. However, relying on this is probably unwise, since future tweaks to the `sigslot` library may change this behaviour.

2.3 Disconnecting Signals

It is very rare to need to explicitly disconnect a signal, because the destructors of both the signal class and the destination class (which must inherit from `has_slots`) automatically disconnect signals as they go out of scope. However, if you need to do so, you can call the signal's `disconnect()` member function with the address of the target class:

```
signal1<int> Bang;
...
Bang.connect(&someobj, &SomeObj::OnBang);
...
Bang(123); // Calls someobj.OnBang()
...
Bang.disconnect(&someobj);
...
Bang(321); // No longer calls someobj.OnBang()
```

2.4 Implementing Slots

Slots are just ordinary member functions, with the following provisos:

1. Slots *must* have a return type of *void*
2. Slots *must* have between 0 and 8 arguments (which can be of any type).
3. Classes implementing slots *must* inherit from `has_slots<>`.

A slot can either be called through the signal/slot mechanism, or it can be called directly as an ordinary member function.

2.5 Completely disconnecting a signal

To disconnect a signal completely from all slots to which it is currently connected, call the signal's `disconnect_all()` member function:

```
signal0<> Bang();

Bang.connect(&bomb, &Bomb::Explode);
Bang.connect(&bomb2, &Bomb::Explode);
Bang.connect(&secret_base, &SecretBase::SelfDestruct);

Bang.disconnect_all();

Bang(); // Safely defused!
```

2.6 Completely disconnecting an object implementing slots

In order to make it easy to completely disconnect an object that implements one or more slots, the `has_slots` base class provides a `disconnect_all()` member function. Calling `disconnect_all()` automatically disconnects *all* connected signals:

[3] Other signal/slot approaches do allow values to be returned, but I found this to be contrived and nasty-looking. So I didn't allow it.

```
class MyClass : public has_slots<>
{
public:
    void OnSpeedChange(double mph);
    void OnBrakesApplied(bool brakestate);
};

MyClass car;

signal1<double> Speed;
signal2<bool> Brakes;

Speed.connect(&car, &MyClass::OnSpeedChange);
Brakes.connect(&car, &MyClass::OnBrakesApplied);

Speed(50.0); // This one gets through
Brakes(true); // So does this

car.disconnect_all();

Speed(31.5); // This one doesn't get through
```

2.7 Emitting an unconnected signal

It is a deliberate design choice that emitting an unconnected signal is *not* an error. Rather, if nothing is connected to a signal, a signal is quietly ignored if it is emitted. No warning is generated, because this is correct behaviour.

The rationale underlying this decision is perhaps not obvious, but in practice this makes certain kinds of application much easier to write.

Consider a reusable class implementing a visual edit control for strings:

```
class StringEdit : public has_slots<>
{
public:
    signal0<> OnReturnPressed;
    signal0<> OnTabPressed;
    signal1<char> OnKeyPressed;
    signal1<char *> OnTextChanged;

    void SetText(char* text); // Slot
    void ClearText(); // Slot

    ...
};
```

Some possible uses of this class might find a use for all of the available signals. However, in many cases, some of the signals would not be useful – therefore, to avoid needing irritating code sequences like[4]

```
if(OnReturnPressed.is_connected())
{
    OnReturnPressed();
}
```

purely to avoid warnings, it makes more sense that simply calling `OnReturnPressed()` should be enough.

[4] Please note – `is_connected()` does not actually exist in the current implementation

3 Implementation Considerations

The sigslot library was written to require only ISO C++ and the C++ Standard Library (std), so it is likely to work unchanged on most platforms, at least in single-threaded mode. Using sigslot in a thread-safe way is currently supported under Win32 and on systems supporting Posix threads (e.g. most Unixes, recent-ish Linux variants, OpenBSD, FreeBSD, Windows 95, 98, ME, NT3.51, NT4.0, Win2k, XP, etc.).

3.1 Library Dependencies

In ISO compliant mode, the current version of sigslot depends only on itself and the Standard Template Library's set and list implementations.

Multithreading support requires the windows.h header under Win32, or pthreads.h under an OS that supports Posix Threads.

3.2 Multithreading Support

The sigslot library currently supports three alternative threading policies:

Single Threaded In single-threaded mode, the library does not attempt to protect its internal data structures across threads. It is therefore essential that all calls to constructors, destructors and signals must exist within a single thread.

Multithreaded Global In multithreaded global mode, the library uses a single, global critical section to protect its internal data structures. This approach has very little overhead in terms of handle usage or memory, but may block unnecessarily sometimes since only a single critical section is shared between all objects.

Multithreaded Local In multithreaded local mode, the library uses a separate critical section per object. This means that every signal has its own critical section, as does every class inheriting from has_slots. These critical sections lock only if absolutely necessary, reducing thread contention in heavily multithreaded applications that heavily use the signal/slot mechanism. However, this comes at a price, because a very much larger number of critical section objects must be created and maintained.

There are two alternative ways to set the threading mode for the library: globally, or on a per-class basis.

All signal classes, and has_slots, take an extra optional parameter which specifies the multithreading policy to be used for that specific class:

```
// Single-threaded
signal1<int, single_threaded> Sig1;

// Multithreaded Global
signal1<int, multi_threaded_global> Sig2;

// Multithreaded Local
signal1<int, multi_threaded_local> Sig3
```

Whilst an application is free to use any combination of threading modes internally, it is a bad idea to combine single and multithreaded policies that need to interoperate with each other. This is one of the only 'violations' that the compiler won't error on, so it is up to the programmer to be careful. Mixing multi_threaded_global and multi_threaded_local is permitted, however, since both properly implement locking semantics.

3.2.1 Setting the threading mode globally

Defining the preprocessor variable SIGSLOT_PURE_ISO forces ISO C++ compliance on all platforms. This turns off thread support, so the threading mode automatically becomes set to single_threaded. If that switch is absent, the library tries to figure out what platform is being used. If WIN32 is defined, Win32 is assumed, and thread support is enabled. Similarly, if __GNUG__ is defined, gcc and Posix Threads are

assumed. Should you be using something other than gcc on a Unix or Unix-like OS, you can define SIGSLOT_USE_POSIX_THREADS to force use of Posix Threads[5].

The default threading mode is set by the SIGSLOT_DEFAULT_MT_POLICY preprocessor variable. This defaults to multi_threaded_global if it is undefined. To set the threading mode globally, make sure that SIGSLOT_DEFAULT_MT_POLICY is set correctly before sigslot.h is included.

The default threading mode is used wherever the threading mode is not explicitly specified - if it is specified, this always overrides the default.

3.2.2 Thread-safety of slots

The sigslot library does not automatically guarantee that your slots will be thread-safe. You should assume that slots can be called at 'inconvenient' times, and should program defensively accordingly.

Whilst sigslot is not intended to be a full-blown threading library, it does include some support that may be useful in making a class that implements slots thread-safe. The has_slots class inherits the multithreading policy, which in turn provides the member functions lock() and unlock(). These functions respectively lock and unlock the mutex[6], and are used to protect the internal data structures used to implement the signal/slot mechanism. It is permissible to use lock() and unlock() in your own code, for example:

```
class MyMultithreadedClass
    : public has_slots<multi_threaded_local>
{
public:
    void Entry1() // Slot
    {
        lock();

        ...

        unlock();
    }

    void Entry2() // Slot
    {
        lock();

        ...

        unlock();
    }
};
```

sigslot provides a useful class which allows the critical section to be automatically locked and unlocked within block scope:

```
class MyMultithreadedClass
    : public has_slots<multi_threaded_local>
{
public:
    void Entry1() // Slot
    {
        lock_block<multi_threaded_local> lock(this);
        ...
    }

    void Entry2()
    {
        lock_block<multi_threaded_local> lock(this);
```

[5] this is necessary because if no known compiler is detected, SIGSLOT_PURE_ISO is assumed.

[6] actually implemented as a critical section under Win32 for performance reasons

```
        ...
    }
};
```

When a `lock_block` object is created, it locks the critical section owned by the passed-in object. When the `lock_block` object goes out of scope, the critical section is automatically released.

3.3 Name Space

The `sigslot` library places all of its definitions within the `sigslot` name space. For reasons of clarity and brevity, the examples in this document all assume that the name space has been opened, e.g.:

```
#include <sigslot.h>
using namespace sigslot;
```

As with the Standard Template Library's `std` name space, it is a matter of personal choice and/or local coding standards as to whether to explicitly use '`sigslot::`' or to open the name space.

3.4 Copy Construction

Full support for copy construction is included, both for copying of `signal` objects and for copying objects that inherit `has_slots`.

There were four possible design choices available when designing the `sigslot` library:

Do nothing, allowing the compiler to generate copy constructors This was not an option, because this will break the internal data structures, resulting in an access violation when a copy of an object is destructed (and probably also some weird behaviour along the way too)

Disable copy construction This could have been achieved by declaring private copy constructors, forcing a compiler error if a copy constructor was accidentally invoked. This is much preferable to the above option, because it does at least prevent nasty things from happening at run time. However, this option would make it impossible to copy-construct any object containing a `signal`, or that inherits `has_slots`. This would be far too restrictive in practice.

Implement dummy copy construction This alternative would implement copy constructors that do nothing. This is better than either of the above approaches, in that it avoids internal problems, but does allow derived classes to be copied. However, any connections would not survive copying, which is inconvenient.

Fully implement copy construction This is the approach taken by the `sigslot` library. If a `signal` is copied, the copy is automatically connected to all of the same slots as the original. Similarly, copying an object that derives from `has_slots` will create an object that is connected to all of the same signals as the original.

Fully implementing copy construction was relatively tricky – it added to the complexity of the library significantly. However, this functionality does make the library usable in the widest possible set of cases, including within STL containers.

3.5 Parameter copying

The function override implementation assumes that parameters are copyable, e.g.:

```
signal1<CErrorStatus> Status;

CErrorStatus err("Broke!");

Status(broke);
```

If the `CErrorStatus` class doesn't support copying, you won't be able to pass it as a parameter. Consider passing a reference instead where possible.

4 Author's Notes

I wrote the `sigslot` library because, after some time programming in the Qt environment, needing to hack MFC code again for a living was somewhat depressing. *I wanted my signals and slots back!* A weekend of recreational programming later, the very first prototype of the library was born. Later, multithreading support for Windows was added, then Posix threads support. Finally, a bit of housekeeping and cleanup work resulted in the library that has now been released.

4.1 Future Enhancements

There are some things I would like to add to the library:

Signal chaining Sometimes it can be useful to be able to *chain* signals together, so that a source signal can be set up to trigger a destination signal (note that I said *signal,* not *slot* here). This avoids the need for slot code that emits a signal then immediately returns. Qt supports this, and from experience I can say that it is quite handy.

Wider platform support I'd like to check out how well *sigslot* sits alongside other existing libraries on as many platforms as possible.

There are some other things I really *don't* want to add to the library, either because I think that they would harm its usefulness, or because I just think that they are too ugly.

Signal Adaptors I strongly believe that signals and slots should type-agree. That is, if a signal has an `int`, a `std::string` and a `double` as its arguments, it *should only ever* be connected to a slot that has exactly matching arguments.

20 years ago, I knew everything and made no mistakes. I always knew exactly what I was doing, and woe betide any compiler that sought to get in my way with its namby-pamby type-checking. 20 years later, I've had enough egg on my face to realise that I do make mistakes – big mistakes are very rare for me, but little slips, typos, getting arguments in the wrong order, missing an argument, that kind of thing, are comparatively frequent. Having gone from BCPL through C to C++, I've also gone from no type-checking whatsoever (BCPL) through moderately lame type checking (C) to pretty rigorous (C++). Strangely enough, through this progression I've spent less and less time prodding debug tools because the compiler catches most of my stupid errors. These days, if my programs compile, they often work first time, or at least after a minimal amount of debugging. I refuse to break this, so I'm not going to add any nasty hacks to `sigslot`.

If you have a signal, and want to connect it to a slot that has different parameters, write some code. That way, the compiler knows exactly what you mean, so if you inadvertently connect the wrong signal to the wrong slot (ask any Qt programmer how many times they've had to track down this kind of thing), the compiler simply won't let you.

I may relent on this issue, if and only if the signal adaptor is explicitly created with code, e.g. through some kind of '`signal_adaptor<...>`' class. I will not support any automation of the creation of such adaptors, however.

Automatically Discarded Parameters This is a variant of the signal adaptor idea, where something like '`signal0<int> fred`' can be conncted to '`void jim()`', with the library dealing with dropping extra arguments automatically. This is technically feasible, but unacceptable due to the large class of possibly accidental usages that could otherwise be flagged by the compiler.

Non-void return values Every attempt at this I've seen has looked weird. If someone can think of a nice, portable, safe and above all clearly readable way to implement this, then by all means propose it. But otherwise, I'm not terribly likely to look favourably on requests that I add this feature.

Connecting signals to global functions I've wrestled with this idea. Some days I like it, some days I don't. It is certainly feasible to implement this. It would definitely complicate the library to a significant extent. My real issue with this is the idea of automatically disconnecting slots when they are no longer valid – when does a global function become valid, and when does it become invalid? Clearly, there is no easy answer to this. A cop-out would be to say that it is the programmer's responsibility, but I am uncomfortable with that. If a slot always resides inside a class, it is easy to know when it is valid – i.e. if an instance of a class exists, its slots are valid. When that instance is

destroyed, its slots become invalid, and are automatically disconnected, thereby making it impossible to call them through the signal/slot mechanism. You just don't get that protection thrown in with a global function.

In a single-threaded application, this isn't really much of an issue – usually, it is easy enough to figure out what can happen when. In a multithreaded environment, as is increasingly common these days, it is far harder to know what interfaces can validly be called and when. The approach currently used by the library has a level of inherent safety that I don't really want to compromise.

So, for now, this feature is off the list.

4.2 Acknowledgements

The original version of `sigslot` was written while I was working at Trayport Computers Ltd. in London[7]. I would like to thank the MD, Eddie Hor, who gave permission for the library to be placed in the public domain. I'd also like to thank James Slaughter, a co-worker and outright C++ guru, for the suggestion that such a library would be possible – this was the spark that made the library happen.

Sarah Thompson
Swindon, UK

[7] http://www.trayport.com/