**INTRODUCTION**

Let's take a look at the example below:



Here we have a dialog box, with a number displayed on it, and a couple buttons which allow you to modify the number. Assume we program it with the MFC (Microsoft Foundation Classes), just simply double click on that dialog, add the message handlers for the 'Increment' and 'Decrement' buttons to update the number in the edit box. From now, we might think: where is the reusability?

A way to do this in MFC, would be inherit from the basic controls (CEditBox and CButton for example), then give them a common interface of some kind. We might want to add a couple of entry points to access and edit the number, 'CEditBox::Increment()' and 'CEditBox::Decrement()', then, the buttons need to implement 'OnClicked()' handlers that call the relevant member functions in CEditBox. And, the instance of CButton must hold the pointer to the exactly the instance of CEditBox that we want to modify.

It's better if we have a general purpose button, that run 'OnClicked()' when we clicked it, and an edit box can be directed to do somethings remotely. The edit box should not need to care what is calling it, and the buttons should not care what they are calling. An unmodified button class can be used to increment, decrement or clear edit box at this time and can be used to remotely turn off my television later.

**THE 'sigslot' LIBRARY**

The sigslot library is a C++ Signal/Slot library. The original version of sigslot library was written by **Sarah Thompson** while working at Trayport Computers Ltd. in London.

Signals and Slots is a powerful mechanism, can be found in Qt framework or Boost C++ libraries, that used for communication between objects. But, we might have licensing requirements problem with Qt or Boost. So, the author created the 'sigslot' library for herself, and for free.

Signal/slot mechanism has a few advantages:

**Syntactically neater.** Signals and slots was made for very readable code. Just define your signals, and define your slots, wire them up, and that's all.

**Inherently robust.** This is the neat part again: when a signal/slot connection is destroyed, the connection is automatically removed. It is therefore impossible to emit a signal and have it arrive at an object that has already been deleted. Best of all, you don't need to write any explicit cleanup code. No problem either the caller goes out of scope or the called goes out of scope.

**Reusability.** The signal/slot mechanism does not require classes to have knowledge of each other, which makes it must easier to develop highly reusable classes.

**SIGNAL/SLOT PARADIGM**

Most of traditional C++ code ultimately packed in number of classes, which inter-operate by calling member functions. Allowing classes to inter-operate this way 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 Light

{

public:

void ToggleState() { std::cout << "Toggle state" << std::endl; }

void TurnOn() { std::cout << "Turn on" << std::endl; }

void TurnOff() { std::cout << "Turn off" << std::endl; }

};

class Switch

{

public:

virtual void Clicked() = 0;

};

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();

}

private:

Light& m\_lp;

};

Light lp1, lp2;

ToggleSwitch tsw1(lp1), tsw2(lp2);

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 Light : public has\_slots<>

{

public:

void ToggleState() { std::cout << "Toggle state" << std::endl; }

void TurnOn() { std::cout << "Turn on" << std::endl; }

void TurnOff() { std::cout << "Turn off" << std::endl; }

};

class Switch

{

public:

signal0<> Clicked;

};

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 all\_on, all\_off;

Light lp1, lp2, lp3, lp4;

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);

**USAGE**

**LIBRARY DEPENDENCIES**

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.

**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.

**NAMESPACE**

The sigslot library places all of its definitions within the sigslot name space.

#include "sigslot\sigslot.h"

using namespace sigslot;

**ARGUMENT TYPES**

The sigslot library can optionally take one or more arguments (but can not higher than 8 arguments). The library has been built upon C++ template mechanism, that means it is fully type-checked.

The naming convention is as follows:

signal***n<***type1, type2, …>

when ***n*** is the number of type arguments.

**CONNECTING SIGNALS WITH SLOTS**

The simple program below contains a window that will emit a signal when the window is resized:

class Window

{

public:

Window() { }

void SetSize(int width, int height)

{

Resized(width, height);

}

public: // signals

signal2<int, int> Resized;

};

class WindowMoniter : public has\_slots<>

{

public: // slots

void OnResize(int width, int height)

{

std::cout << width << " x " << height << std::endl;

}

};

Window w;

WindowMoniter m;

w.Resized.connect(&m, &WindowMoniter::OnResize);

Remember that the types of signals and slots need to agree in order for a connection to be made - the used names do not matter.

In the program above, after connected signal 'Resized' of Window object to slot 'OnResize' of WindowMonitor object, each time we set new size to the window, the slot 'OnResize' is automatically called with the 'width' and 'height' arguments passed from the signal.

**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:

Window w;

WindowMoniter m;

w.Resized.connect(&m, &WindowMoniter::OnResize);

w.Resized.disconnect(&m);

**COMPLETELY DISCONNECTING A SIGNALS**

To disconnect a signal completely from all slots to which it is currently connected, just simply call the signal 'disconnect\_all()' member function:

Window w;

WindowMoniter m1, m2, m3;

w.Resized.connect(&m1, &WindowMoniter::OnResize);

w.Resized.connect(&m2, &WindowMoniter::OnResize);

w.Resized.connect(&m3, &WindowMoniter::OnResize);

w.Resized.disconnect\_all();

w.SetSize(1024, 768); // connection terminated

**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:

class Car : public has\_slots<>

{

public: // slots

void OnSpeedChanged(float speed) { }

void OnDoorLocked(bool locked) { }

};

Car c;

signal1<float> Speed;

signal1<bool> Locker;

Speed.connect(&c, &Car::OnSpeedChanged); // change speed

Locker.connect(&c, &Car::OnDoorLocked); // lock the doors

c.disconnect\_all(); // remove all control

Speed(50.5); // This one can not be effected

**EMITTING AN UNCONNECTED SIGNAL**

Emitting an unconnected signal is not an error. If nothing is connected to a signal, that signal is quitely ignored if it is emitted. No warning is generated, because this is correct behaviour.