libsigc++

Ainsley Pereira

COLLABORATORS			
	TITLE : libsigc++		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Ainsley Pereira	September 2002	

Contents

Abstract

libsigc++ is a C++ template library implementing typesafe callbacks. This is an intro to libsigc++.

Chapter 1

Introduction

1.1 Motivation

There are many situations in which it is desirable to decouple code that detects an event, and the code that deals with it. This is especially common in GUI programming, where a toolkit might provide user interface elements such as clickable buttons but, being a generic toolkit, doesn't know how an individual application using that toolkit should handle the user clicking on it.

In C the callbacks are generally handled by the application calling a 'register' function and passing a pointer to a function and a `void*` argument, eg.

```
void clicked(void* data);

button* okbutton = create_button("ok");
static char somedata[] = "This is some data I want the clicked() function  ↩
    to have";

register_click_handler(okbutton, clicked, somedata);
```

When clicked, the toolkit will call `clicked()` with the data pointer passed to the `register_click_handler()` function.

This works in C, but is not typesafe. There is no compile-time way of ensuring that `clicked()` isn't expecting a struct of some sort instead of a `char*`.

As C++ programmers, we want type safety. We also want to be able to use things other than free-standing functions as callbacks.

libsigc++ provides the concept of a slot, which holds a reference to one of the things that can be used as a callback:

- A free-standing function as in the example
- A functor object that defines operator() (a lambda expression is such an object)
- A pointer-to-a-member-function and an instance of an object on which to invoke it (the object should inherit from `sigc::trackable`)

All of which can take different numbers and types of arguments.

To make it easier to construct these, libsigc++ provides the sigc::ptr_fun() and sigc::mem_fun() functions, for creating slots from static functions and member functions, respectively. They return a generic `signal::slot` type that can be invoked with `emit()` or `operator()`.

For the other side of the fence, libsigc++ provides `signal`s, to which the client can attach `slot`s. When the `signal` is emitted, all the connected `slot`s are called.

Chapter 2

Connecting your code to signals

2.1 A simple example

So to get some experience, lets look at a simple example...

Lets say you and I are writing an application which informs the user when aliens land in the car park. To keep the design nice and clean, and allow for maximum portability to different interfaces, we decide to use libsigc++ to split the project in two parts.

I will write the `AlienDetector` class, and you will write the code to inform the user. (Well, OK, I'll write both, but we're pretending, remember?)

Here's my class:

```
class AlienDetector
{
public:
    AlienDetector();

    void run();

    sigc::signal<void()> signal_detected;
};
```

(I'll explain the type of signal_detected later.)

Here's your code that uses it:

```
void warn_people()
{
    std::cout << "There are aliens in the carpark!" << std::endl;
}

int main()
{
    AlienDetector mydetector;
    mydetector.signal_detected.connect( sigc::ptr_fun(warn_people) );

    mydetector.run();

    return 0;
}
```

You can use a lambda expression instead of sigc::ptr_fun().

```
    mydetector.signal_detected.connect( [](){ warn_people(); } );
```

Pretty simple really - you call the `connect()` method on the signal to connect your function. `connect()` takes a `slot` parameter (remember slots are capable of holding any type of callback), so you convert your `warn_people()` function to a slot using the `slot()` function.

To compile this example, use:

```
g++ example1.cc -o example1 `pkg-config --cflags --libs sigc++-2.0`
```

Note that those `` characters are backticks, not single quotes. Run it with

```
./example1
```

(Try not to panic when the aliens land!)

2.2 Using a member function

Suppose you found a more sophisticated alien alerter class on the web, such as this:

```
class AlienAlerter : public sigc::trackable
{
public:
    AlienAlerter(char const* servername);
    void alert();
private:
    // ...
};
```

(Handily it derives from `sigc::trackable` already. This isn't quite so unlikely as you might think; all appropriate bits of the popular gtkmm library do so, for example.)

You could rewrite your code as follows:

```
int main()
{
    AlienDetector mydetector;
    AlienAlerter  myalerter("localhost"); // added
    mydetector.signal_detected.connect( sigc::mem_fun(myalerter, & ↩
        AlienAlerter::alert) ); // changed

    mydetector.run();

    return 0;
}
```

Note that only 2 lines are different - one to create an instance of the class, and the line to connect the method to the signal.

This code is in example2.cc, which can be compiled in the same way as example1.cc

It's possible to use a lambda expression instead of sigc::mem_fun(), but it's not recommended, if the class derives from `sigc::trackable`. With a lambda expression you would lose the automatic disconnection that the combination of `sigc::trackable` and sigc::mem_fun() offers.

2.3 Signals with parameters

Functions taking no parameters and returning void are quite useful, especially when they're members of classes that can store unlimited amounts of safely typed data, but they're not sufficient for everything.

What if aliens don't land in the carpark, but somewhere else? Let's modify the example so that the callback function takes a `std::string` with the location in which aliens were detected.

I change my class to:

```
class AlienDetector
{
public:
    AlienDetector();

    void run();

    sigc::signal<void(std::string)> signal_detected;  // changed
};
```

The only line I had to change was the signal line (in `run()` I need to change my code to supply the argument when I emit the signal too, but that's not shown here).

The name of the type is '`sigc::signal`'. The template parameters are the return type, then the argument types in parentheses. (libsigc++2 also accepts a different syntax, with a comma between the return type and the parameter types. That syntax is deprecated, though.)

The types in the function signature are in the same order as the template parameters, eg:

```
sigc::signal<void(std::string)>
    void function(std::string foo);
```

So now you can update your alerter (for simplicity, lets go back to the free-standing function version):

```
void warn_people(std::string where)
{
    std::cout << "There are aliens in " << where << "!" << std::endl;
}

int main()
{
    AlienDetector mydetector;
    mydetector.signal_detected.connect( sigc::ptr_fun(warn_people) );

    mydetector.run();

    return 0;
}
```

Easy.

2.4 Disconnecting

If you decide you no longer want your code to be called whenever a signal is emitted, you must remember the return value of `connect()`, which we've been ignoring until now.

`connect()` returns a `sigc::connection` object, which has a `disconnect()` member method. This does just what you think it does.

Chapter 3

Writing your own signals

3.1 Quick recap

If all you want to do is use gtkmm, and connect your functionality to its signals, you can probably stop reading here.

You might benefit from reading on anyway though, as this section is going to be quite simple, and the 'Rebinding' technique from the next section is occasionally useful.

We've already covered the way the types of signals are made up, but lets recap:

A signal is an instance of a template, named `sigc::signal`. The template arguments are the types, in the order they appear in the function signature that can be connected to that signal; that is the return type, then the argument types in parentheses.

To provide a signal for people to connect to, you must make available an instance of that `sigc::signal`. In `AlienDetector` this was done with a public data member. That's not considered good practice usually, so you might want to consider making a member function that returns the signal by reference. (This is what gtkmm does.)

Once you've done this, all you have to do is emit the signal when you're ready. Look at the code for `AlienDetector::run()`:

```
void AlienDetector::run()
{
    sleep(3); // wait for aliens
    signal_detected.emit(); // panic!
}
```

As a shortcut, `sigc::signal` defines `operator()` as a synonym for `emit()`, so you could just write `signal_detected();` as in the second example version:

```
void AlienDetector::run()
{
    sleep(3);                // wait for aliens
    signal_detected("the carpark"); // this is the std::string version,  ↩
        looks like
                             // they landed in the carpark after all.
}
```

3.2 What about return values?

If you only ever have one slot connected to a signal, or if you only care about the return value of the last registered one, it's quite straightforward:

```
sigc::signal<int()> somesignal;
int a_return_value;

a_return_value = somesignal.emit();
```

Chapter 4

Advanced topics

4.1 Rebinding

Suppose you already have a function that you want to be called when a signal is emitted, but it takes the wrong argument types. For example, lets try to attach the `warn_people(std::string)` function to the detected signal from the first example, which didn't supply a location string.

Just trying to connect it with:

```
myaliendetector.signal_detected.connect(sigc::ptr_fun(warn_people));
```

results in a compile-time error, because the types don't match. This is good! This is typesafety at work. In the C way of doing things, this would have just died at runtime after trying to print a random bit of memory as the location - ick!

We have to make up a location string, and bind it to the function, so that when signal_detected is emitted with no arguments, something adds it in before `warn_people` is actually called.

We could write it ourselves - it's not hard:

```
void warn_people_wrapper() // note this is the signature that ' ↩
    signal_detected' expects
{
    warn_people("the carpark");
}
```

but after our first million or so we might start looking for a better way. As it happens, libsigc++ has one.

```
sigc::bind(slot, arg);
```

binds arg as the argument to slot, and returns a new slot of the same return type, but with one fewer arguments.

Now we can write:

```
myaliendetector.signal_detected.connect(sigc::bind( sigc::ptr_fun( ↩
    warn_people), "the carpark" ) );
```

If the input slot has multiple args, the rightmost one is bound.

The return type can also be bound with `sigc::bind_return(slot, returnvalue);` though this is not so commonly useful.

So if we can attach the new `warn_people()` to the old detector, can we attach the old `warn_people` (the one that didn't take an argument) to the new detector?

Of course, we just need to hide the extra argument. This can be done with `sigc::hide`, eg.

```
myaliendetector.signal_detected.connect( sigc::hide<std::string>( sigc:: ↩
    ptr_fun(warn_people) ) );
```

The template arguments are the types to hide (from the right only - you can't hide the first argument of 3, for example, only the last).

`sigc::hide_return` effectively makes the return type void.

4.2 Retyping

A similar topic is retyping. Perhaps you have a signal that takes an `int`, but you want to connect a function that takes a `double`.

This can be achieved with the `sigc::retype()` template. It takes a `sigc::slot`, and returns a `sigc::slot`. eg.

```
void dostuff(double foo)
{
}

sigc::signal<void(int)> asignal;

asignal.connect( sigc::retype( sigc::ptr_fun(&dostuff) ) );
```

If you only want to change the return type, you can use `sigc::retype_return()`. `retype_return()` needs one template argument, the new return type.

Chapter 5

Reference

See the reference documentation online