Callbacks in C++

Ted Felix

Callbacks are tremendously useful in object-oriented design when one needs to decouple two classes but let them be connected via a single function call. Examples abound and at some point I'll expand. For now, this is a place for me to keep my experiments with various approaches to implementing callbacks in C++.

Ideal Solution

Wouldn't it be nice if C++ supported callbacks directly? Let's say we have a timer class that takes a callback which it will call periodically. It would be great to be able to do this:

```
Timer t;  // caller
MyClass myObject;  // callee

t.connect(myObject.foo);  // callback
t.start(1);  // 1 second period
```

Every second, the Timer object `t` would call the connected callback function `myObject.foo()`.

Unfortunately, C++ does not offer anything like this. `myObject.foo` is a combination of the object pointer `&myObject` and the member function pointer `foo`. C++ does not have a pointer type that combines these two pointers.

Interface Class

One technique for implementing callbacks is to use an interface class. An interface class contains a member function to be overridden by a deriver. This is the callback function.

```
class CallbackInterface
{
public:
    // The prefix "cbi" is to prevent naming clashes.
    virtual int cbiCallbackFunction(int) = 0;
};
```

The class that wants to be called back derives from the CallbackInterface and implements the callback function.

```
class Callee : public CallbackInterface
{
public:
    // The callback function that Caller will call.
    int cbiCallbackFunction(int i)
    {
        printf("  Callee::cbiCallbackFunction() inside callback\n");
        return 2 * i;
    }
};
```

Now a pointer to callee can be passed to a function or object that will call it back.

```
Caller caller;
Callee callee;

// Connect the callback
caller.connectCallback(&callee);

// Test the callback
caller.test();
```

And now within Caller, we can store the callback (as a CallbackInterface*) and call it anytime we need to.

```
class Caller
{
public:
    // Clients can connect their callback with this
    void connectCallback(CallbackInterface *cb)
    {
        m_cb = cb;
    }

    // Test the callback to make sure it works.
    void test()
    {
        printf("Caller::test() calling callback...\n");
        int i = m_cb -> cbiCallbackFunction(10);

        printf("Result (20): %d\n", i);
    }

private:
    // The callback provided by the client via connectCallback().
    CallbackInterface *m_cb;
};
```

This approach takes advantage of inheritance and polymorphism to implement a callback. This works especially well if the callback interface requires more than one function. The problem with this approach is when we want to connect a single callee to a number of instances of the same caller. As an example, if we want to create three or four timers and handle each one in a different callback function (e.g. onOneSecondTimer(), onTwoSecondTimer(), ...). Since the interface class forces us to use a specific function name, we cannot do this.

callback1.cpp - Complete example.

Function Pointer

In C, function pointers are the easiest way to implement callbacks and they can be made to work in a C++ class, although it is a little awkward. We'll need two functions to pull this off. The first is a static callback function, and the second is a member callback function.

```
class Callee
```

```
{
public:
    // This static function is the real callback function.  It's compatible
    // with the C-style CallbackFunctionPtr.  The extra void* is used to
    // get back into the real object of this class type.
    static int staticCallbackFunction(void *p, int i)
    {
        // Get back into the class by treating p as the "this" pointer.
        ((Callee *)p) -> callbackFunction(i);
    }

    // The callback function that Caller will call via
    // staticCallbackFunction() above.
    int callbackFunction(int i)
    {
        printf("  Inside callback\n");
        return 2 * i;
    }
};
```

The trick here is that staticCallbackFunction() assumes that "p" is a pointer to the Callee object. It uses that to get back into the object and call the member function callbackFunction. In main():

```
Caller caller;
Callee callee;

// Connect the callback.  Send the "this" pointer for callee as the
// void* parameter.
caller.connectCallback(Callee::staticCallbackFunction, &callee);

// Test the callback
caller.test();
```

So, we send both the pointer to the function (Callee::staticCallbackFunction) and the "this" pointer for the object (&callee) to the class that will do the calling back. This is everything we need to get back in.

Caller just needs to store the "this" pointer and the function pointer. It can then use them when doing the callback.

```
typedef int(*CallbackFunctionPtr)(void*, int);

class Caller
{
public:
    // Clients can connect their callback with this.  They can provide
    // an extra pointer value which will be included when they are called.
    void connectCallback(CallbackFunctionPtr cb, void *p)
    {
        m_cb = cb;
        m_p = p;
    }

    // Test the callback to make sure it works.
    void test()
    {
        printf("Caller::test() calling callback...\n");
        int i = m_cb(m_p, 10);
```

```
            printf("Result (20): %d\n", i);
        }

    private:
        // The callback provided by the client via connectCallback().
        CallbackFunctionPtr m_cb;
        // The additional pointer they provided (it's "this").
        void *m_p;
    };
```

For use in C++ code, this is a fairly cumbersome approach. You've got to define two functions for every callback: the static function and the actual callback function. When interfacing with C code that uses function pointers for callbacks, this is a perfect approach. It successfully makes the jump from C to C++.

callback3.cpp - Full example.

TODO: Investigate making the void something more typesafe. Or using dynamic_cast<>. Actually, the void* is for compatibility with C APIs, so perhaps that should be mentioned.*

C++11 Lambda Functions

C++11 introduces many handy new features to the language. One of them is lambda functions. A lambda function is an anonymous, temporary, usually small function. By combining the new std::function with a lambda function that looks very similar to the above function pointer approach, we can have a pretty decent callback mechanism.

The Callee in this case is a normal member function. Note that I've thrown in a member m_i to show that the "this" pointer is indeed correct.

```
    class Callee
    {
    public:
        Callee(int i) : m_i(i) { }

        // The callback function that Caller will call.
        int callbackFunction(int i)
        {
            printf("  Callee::callbackFunction() inside callback\n");
            return m_i * i;
        }

    private:
        // To prove "this" is indeed valid within callbackFunction().
        int m_i;
    };
```

In main(), we see the connection being made with a lambda. Like the static function in the C function pointer approach, the lambda captures the "this" pointer so that it can get into the class. The lambda syntax is a tad obtuse. This is probably the only real drawback to this approach.

```
    Caller caller;
    Callee callee(5);
```

```
// Connect the callback.  Like with the C-style function pointer and
// static function, we use a lambda to get back into the object.
caller.connectCallback(
    [&callee](int i) { return callee.callbackFunction(i); });

// Test the callback
caller.test();
```

If you aren't familiar with lambda functions in C++, here's a quick breakdown. "[&callee]" says to "capture" callee by reference. This basically means to make the variable "callee" available within the lambda function. We do this by reference to avoid a copy which would be disastrous. "(int i)" says that our lambda function takes a single int parameter. This matches callbackFunction(). Within the braces is the code for the lambda function. In this case we simply delegate the call to callbackFunction(). Note that we do not specify the return type of our lambda function because C++11 is clever and will figure it out automatically.

The caller uses the straightforward std::function to store the callback and the actual call is very simple.

```
typedef std::function<int(int)> CallbackFunction;

class Caller
{
public:
    // Clients can connect their callback with this.
    void connectCallback(CallbackFunction cb)
    {
        m_cb = cb;
    }

    // Test the callback to make sure it works.
    void test()
    {
        printf("Caller::test() calling callback...\n");
        int i = m_cb(10);

        printf("Result (50): %d\n", i);
    }

private:
    // The callback provided by the client via connectCallback().
    CallbackFunction m_cb;
};
```

This is so close to perfection. The only drawback is that the lambda function syntax is slightly bizarre. Other than that, the rest of this is pretty much perfect.

callback4.cpp

Template Functors (Rich Hickey)

I would say that Rich Hickey's template functor callback approach is slightly better than C++11's lambda functions. It's more direct and closer to the ideal solution. It also works with pre-C++11 compilers. I've used this for many years and recommend it.

There are two key differences between the lambda functions and Rich Hickey's template functor approach. First, instead of using std::function, we need to use the slightly more clunky CBFunctor* types:

```
// C++11 lambda version
typedef std::function<int(int)> CallbackFunction;

// Rich Hickey version
typedef CBFunctor1wRet<int, int> CallbackFunction;
```

Lastly, we use makeFunctor() to create the callback. This is somewhat easier to understand than the lambda syntax.

```
// C++11 lambda version
caller.connectCallback([&callee](int i) { return callee.foo(i); });

// Rich Hickey version
caller.connectCallback(makeFunctor(callee, &Callee::foo));
```

Note that the above makeFunctor() does not have the extra first parameter to differentiate between the functor types. I'll let Rich explain from his article: "I must come clean at this point, and point out that the syntax above for makeFunctor() is possible only in the proposed language, because it requires template members (specifically, the Functor constructors would have to be templates). In the current language the same result can be achieved by passing to makeFunctor() a dummy parameter of type ptr-to-the-Functor-type-you-want-to-create. This iteration of the callback library requires you pass makeFunctor() the dummy as the first parameter. Simply cast 0 to provide this argument." My current example code has this dummy parameter as I've not yet found a version of this callback header that supports the above syntax. A search should turn up a version of the header that can handle this. I'm pretty sure I've seen them go by. (Template members are covered on pg 672 of Lippman 2013 if you want to give it a shot. Might be worth some fiddling.)

TODO: I seriously need to hunt down a more modern implementation of Template Functors that has the simpler makeFunctor() argument list. Then I need to clean it up and modernize it. Perhaps put it up on sourceforge or github. Oh, and try to re-create the .cpp file too. A test suite would be nice.

callback5.cpp
callback.h

Callbacks in C++ Using Template Functors (Rich Hickey 1994) - Covers the interface class approach (Callee Mix-In) and the function pointer approach (Function Model).

More?

There are other C++ callback libraries out there. I need to look around a bit and pull some of them into here for analysis. A Google search should turn them up pretty quickly.

License

<- Back to my software page.