

Polymorphic Function Objects

by Stephen C. Dewhurst

C++ programmers are a loyal bunch. Whenever we're faced with a situation that requires a feature the language doesn't support, we don't abandon C++ to flirt with some other language; we just extend C++ to support the feature in which we're interested.

C++ is designed for this type of extension. Not only does it support the usual basic requirements of any language that allows the creation of abstract data types (encapsulation and access protection), it also provides the ability to overload operators, and to define initialization, destruction, copy, and conversion semantics.

It is also important that C++ (and, presumably, its creator Bjarne Stroustrup) elects *not* to push a fixed philosophy of what constitutes a well-written program or a good design. These issues have wisely been left to the community of C++ programmers as a whole. C++ is a relatively complex language, and this complexity is mitigated by the availability of C++ programming idioms and somewhat higher-level design patterns that describe commonly accepted styles of implementing certain low and mid-level designs. Since they are socially determined, these idioms evolve over time so as to be in sync with the current needs of the C++ programming community. Less complex and less flexible languages, or languages that come with a significant amount of "proper design" baggage, are invariably replaced in short order by new languages that are more suitable to actual needs.

In this installment, we'll look at the C++ "function object" idiom. As with other areas of the base language--like pointers, arrays, and numeric types--we often find a need to extend the base language to support a function or function pointer with augmented functionality.

Calling Back

Suppose you're taking a long trip and I lend you my car for the purpose. Given the state of my car, I'll probably also hand you a sealed envelope with a telephone number in it, along with the instructions to call the number in the envelope if you experience any engine problems. This is a callback. You do not have to know the number in advance (it might be the number of a good repair shop, a bus line, or the city dump), and in fact you may never have to call the number. In effect, the task of handling the "engine trouble" event has been partitioned between you (also known as the framework) and me (also known as the client of the framework). You know when it's time to do something, but not what to do. I know what to do if a particular event occurs, but not when to do it. Together we make an application.

A more prosaic example of the use of a callback than an exploding internal combustion engine might be a GUI button.

```
typedef void (*Action)();
```

```
class Button {
public:
    Button( const char *label );
    ~Button();
    void press() const;
    void setAction( Action );
private:
    string _label;
    Action _action;
};

Button::Button( const char *label )
    : _label( label ), _action( 0 ) {}

void Button::setAction( Action action )
    { _action = action; }

void Button::press() const
    { if( _action ) _action(); }
```

Here we've employed the traditional approach of using a pointer to function as our callback type. This decouples the framework (button) code from the client of the framework, and allows the same button to be reconfigured with different callbacks at different times. (For example, the owner of the first repair shop you call may, after fixing the engine, give you a sealed envelope containing the number of another repair shop further down the road--owned by his brother.)

```
Button doit( "Go ahead, press me" );
extern void abort();
doit.setAction( abort );
//...
if( theMoodStrikes() )
    doit.press();
//...
extern void sendFlowers();
doit.setAction( sendFlowers );
```

The `sendFlowers` function shows one weakness with this approach. Presumably we'd like to send flowers to a particular delivery location. Functions often need data. Unfortunately, our button expects to call back on a function of a particular type--one that takes no argument and returns `void`--so we have to find another way to make the delivery location available to our function. How about a global variable?

```
extern Location deliveryLocation;
//...
deliveryLocation = myHouse;
doit.setAction( sendFlowers );
```

This will work for a while, until a summer intern gets ahold of the code and modifies it to perform additional actions, or uses the same function simultaneously as a callback of two different buttons. Unfortunately, there is no sure way to bind a global variable to a specific function, and the intern may (for reasons that I have never been able to fathom) decide to reuse a global variable for a new purpose. Like flower deliveries, missiles often have delivery locations as well.

```
extern void launchMissile(); // ->deliveryLocation
doit.setAction( launchMissile ); // oops!
```

Another approach is to rewrite our button class to permit a more flexible callback function.

```
class Button2 {
    //...
    void setAction( void (*action)( void * ) );
    void setArg( void *arg );
    void press() const
        { if( _action ) _action( _arg ); }
private:
    //...
    void (*_action)( void * );
    void *_arg;
};
```

Too flexible. Here we give up any semblance of type checking by passing a pointer to untyped data, that then must be provided with a type by the callback function, typically through use of a cast. If the callback function guesses the data's type incorrectly, we have a runtime type error.

Function Object Callbacks

Let's see...the problem is that we have to bind a function to the data with which it works in such a way that we can deal with them as a unit in a type-safe manner. A class. Let's rewrite the button type to use a function object as a callback.

```
class Action {
public:
    virtual ~Action();
    virtual void operator () () = 0;
};
```

```
class Button {
public:
    Button( const char *label );
    ~Button();
    void press() const;
    void setAction( Action & );
private:
    string _label;
    Action *_action; // doesn't own
};

Button::Button( const char *label )
    : _label( label ), _action( 0 ) {}

void Button::setAction( Action &action )
    { _action = &action; }

void Button::press() const
    { if( _action ) (*_action)(); }
```

The `Action` class is the base of a polymorphic function object hierarchy, and the customization of `Button` with such an object is an instance of the Command pattern.¹ In this article we'll concentrate on function object hierarchies as the bases of flexible runtime customization. Other uses of non-polymorphic function objects, notably in the STL, form the bases for efficient and flexible compile-time customization.

Note that we have chosen to overload the function call operator in the `Action` class to document, implicitly, that `Action` is a class that is to be used in place of a function or function pointer. In essence, an `Action` is a “smart function” in a way similar to the perhaps more common use of classes that overload the arrow operator as “smart pointers.”

Now we can safely bind a callback with any required data in a type-safe way.

```
class SendFlowers : public Action {
public:
    SendFlowers( const Location & );
    void operator () ();
private:
    const Location &_deliveryLocation;
};
```

¹ Gamma, et al., *Design Patterns*, Addison Wesley 1995, p. 233.

```
class LaunchMissile : public Action {
public:
    LaunchMissile( Location & );
    void operator () ();
private:
    Location &_deliveryLocation;
};
struct Abort : public Action {
    void operator () ()
        { abort(); }
};
//...
SendFlowers sendMeFlowers( myHouse );
doit.setAction( sendMeFlowers );

LaunchMissile launchTest( testLoc );
doit.setAction( launchTest );
```

Function Objects are Objects

One of the major benefits of function objects is that they are, well, objects. As instances of classes, they allow us to leverage our other techniques for manipulating and composing class objects.

We may want to have a button execute a sequence of commands when pressed. I, personally, always send flowers to a delivery location before launching a missile. It's the least one can do. I could hand code a "send-flowers-then-launch" function object, but I'd rather avoid the effort. Instead, I'll allow an Action to be a Composite.²

```
class Macro : public Action {
public:
    ~Macro();
    void add( Action *a )
        { _actions.push_back( a ); }
    void operator () () {
        for( list<Action *>::iterator i(_a.begin());
            i != _a.end(); ++i )
            (**i) ();
    }
```

² Ibid., p. 163.

```

    }
private:
    list<Action *> _a;
};

```

I don't know about you, but I sometimes like to beep after executing a callback. I also get the occasional urge to record callback results in a log file. Unfortunately, producing custom Actions like `LaunchMissile`, `LaunchMissileBeep`, `LaunchMissileLog`, and `LaunchMissileLogBeep` can be tedious. Instead, I'll allow an Action to be decorated.³

```

class ActionDec : public Action {
public:
    ActionDec( Action *toDecorate ) : a_( toDecorate ) {}
    void operator ()() = 0;
private:
    Action *a_;
};

```

```

void ActionDec::operator ()()
{ (*a_)(); }

```

```

class Beeper : public ActionDec {
public:
    Beeper( Action *toDecorate ) : ActionDec( toDecorate ) {}
    void operator()() {
        ActionDec::operator()(); // forward call
        cout << '\a' << flush; // augment action
    }
};

```

I also run up against situations where I recognize that I want to attach the same callback to different contexts, but allow them to remain independent. Unfortunately, I don't always know the specifics of the callback; after all, it could be a beeping launching logging efflorescent abort. Let's apply the Prototype⁴ pattern to allow cloning of Actions.

```

class Action {
public:
    Action();

```

³ Ibid., p. 175.

⁴ Ibid., p. 117.

```
virtual ~Action();
virtual void operator () () = 0;
virtual Action *clone() const = 0;
};

struct Abort : public Action {
    void operator () () { abort(); }
    Abort *clone() const
        { return new Abort; }
};

struct NullAction : public Action {
    void operator () () {}
    NullAction *clone() const
        { return new NullAction; }
};

class Macro : public Action {
public:
    Macro();
    ~Macro();
    void add( Action *a );
    void operator () ();
    Macro *clone() const;
private:
    list<Action *> _a;
};

Macro *Macro::clone() const {
    Macro *m = new Macro;
    for( list<Action *>::const_iterator i( _a.begin());
        i != _a.end(); ++i )
        m->add( (*i)->clone() );
    return m;
}
```

Our previous version of the `Button` class was unsafe in that it made the tenuous assumption that the `Action` passed to the `setAction` member function would remain in existence at least as long as the

Button, and that no other contexts were using that Action in a way that would adversely affect the behavior of the Button. A much safer alternative would be to set a Button's callback to a copy of the argument to setAction. The clone function allows us to do this while remaining happily ignorant of the details of the Action we copy.

```
class Button {
public:
    Button( const char *label );
    ~Button();
    void press() const;
    void setAction( const Action * );
    const Action *getAction() const;
private:
    string _label;
    Action *_action;
};

Button::Button( const char *label )
    : _label( label ), _action( new NullAction ) {}

Button::~~Button()
    { delete _action; }

void Button::setAction( const Action *action ) {
    if( action ) {
        delete _action;
        _action = action->clone();
    }
}
```

Members Functions Want to be Objects Too

Enough callbacks. Let's do some numerical integration. Our first attempt allows us to integrate a non-member function.

```
class Quad {
public:
    Quad( double (*func)( double ) );
    ~Quad();
```

```
    double integrate( double low, double high );
private:
    double (*_theFunc)( double );
};
```

We initialize a `Quad` with a function (or function pointer) and integrate between two endpoints. The implementation of the `integrate` function probably invokes the function repeatedly to calculate and sum the areas of a sequence of rectangles or trapezoids between the endpoints. We assume that the function is both integrable and “reasonable” (connected, not too spiky, etc.)

```
extern double aFunction( double );
Quad quad( aFunction );
double area = quad.integrate( 0.0, 2.71828 );
```

Unfortunately, we cannot use `Quad` to integrate a member function. This is a shame, since although pointers to member function are functionally very different from pointers to nonmember functions, they are logically quite similar. To allow integration of both member and nonmember functions we’ll resort, once again, to a polymorphic function object.

```
class Func {
public:
    virtual ~Func();
    virtual double operator ()( double ) = 0;
};
class Quad {
public:
    Quad( Func & );
    ~Quad();
    double integrate( double low, double high );
    //...
};
```

The nonmember function type is straightforward.

```
class NMFunc : public Func {
public:
    NMFunc( double (*func)( double ) );
    ~NMFunc();
    double operator ()( double );
private:
    double (*_func)( double );
};
```

```
double
NMFunc::operator ()( double d )
    { return _func( d ); }
```

Integration of a nonmember function is similar to the original version of `Quad` (albeit somewhat slower).

```
NMFunc f1( aFunction );
Quad quad( f1 );
double area = quad.integrate( 0.0, 2.71828 );
```

Member functions are handled by binding a pointer to member function with a class object used to dereference it. We use a template to allow integration over member functions of different classes.

```
template <class C>
class MFunc : public Func {
public:
    MFunc( C &obj, double (C::*func)(double) );
    ~MFunc();
    double operator ()( double );
private:
    C &_obj;
    double (C::*_func)( double );
};
```

```
template <class C>
double
MFunc<C>::operator ()( double d )
    { return (_obj.*_func)( d ); }
```

Integration of a member function is similar to that of a nonmember function.

```
X xobj;
MFunc<X> f2( xobj, &X::aMemberFunc );
Quad quad( f2 );
double area = quad.integrate( 0.0, 2.71828 );
```

Functional Observations

I like to use callbacks to motivate function objects for a number of reasons. First, “everyone knows” how to implement a callback--with a function pointer--and “everyone” is wrong. In applying a simple reification of a function into an object we are able to overcome the obvious deficits of function pointers in a simple and straightforward way.

The real benefit of the reification, however, lies in our ability to plug our function objects into the

generative power afforded by many of the design patterns and C++ idioms defined for objects. Like most successful complex designs, our `Action` hierarchy is the result of the interaction of a small number of simple designs. This allows the hierarchy to be described simply, understood readily, and maintained and modified easily.

Another benefit of the approach of composing pattern and idiom is that often these techniques can be applied independently and in ignorance of each other. As the purveyor of a framework, we may provide only the `Button` and `Action` classes. Users of the framework complete the application by providing concrete `Actions`. Some of these actions may compose other actions with a `Composite`, or augment them with a `Decorator`. If both composite and decorated actions happen to be present in a particular customization of our framework, we can compose beeping launching logging efflorescent aborts.

The reification afforded by the use of function objects also allows us to unify what are really very different parts of the C++ language under a common abstraction. In the case of the numerical integration we examined above, we wanted to treat both pointers to nonmember functions and pointers to member functions as essentially the same. Use of a polymorphic function object hierarchy allowed us to achieve this unification easily and effectively.

end of article