Item 18. Function Objects from C++ Common Knowledge

Often you'll need something that behaves like a function pointer, but function pointers tend to be unwieldy, dangerous, and (let's admit it) passé. Often the best approach is to use a function object instead of a function pointer.

A function object, like a smart pointer (see Smart Pointers [42, 145]) is an ordinary class object. Whereas a smart pointer type overloads the -> and * (and possibly ->*) operators to mimic a "pointer on steroids," a function object type overloads the function call operator, (), to create a "function pointer on steroids." Consider the following function object that computes the next element in the well-known Fibonacci series (1, 1, 2, 3, 5, 8, 13, ...) with each call:

class Fib {
    public:
        Fib() : a0_(1), a1_(1) {}
        int operator (){
            private:
                int a0_, a1_;
            }
        int Fib::operator () {
            int temp = a0_;  
            a0_ = a1_; 
            a1_ = temp + a0_; 
            return temp; 
        }
}

A function object is just a regular class object, but you can call its operator () member (or members, if there is more than one) with standard function call syntax.

Fib fib;
//...
cout << "next two in series: " << fib() << " " << fib() << endl;

The syntax fib() is recognized by the compiler as a member function call to the operator () member of fib, identical in meaning to fib.operator() but presumably easier on the eye. The advantage in this case of using a function object in preference to a function or a pointer to a function is that the state required to compute the next number in the Fibonacci series is stored within the Fib object itself. A function implementation would have to resort to global or local static variables or some other base trickery to retain state between invocations of the function, or the information would have to be passed to the function explicitly. Also note that unlike a function that uses static data, we can have multiple, simultaneous Fib objects whose calculations do not interfere with each other.

int fibonacci () {
    static int a0 = 0, a1 = 1; // problematic...
    int temp = a0;
It's also possible and common to create the effect of a virtual function pointer by creating a function object hierarchy with a virtual operator (). Consider a numeric integration facility that calculates an approximation of the area under a curve, as shown in Figure 5.

**Figure 5.** Numeric integration by summing areas of rectangles (simplified)

[View full size image]

Our integration function will iteratively call a function for values between low and high in order to approximate the area under the curve as a sum of the areas of rectangles (or some similar mechanism).

```cpp
typedef double (*F)( double );
double integrate( F f, double low, double high ) {
    const int numsteps = 8;
    double step = (high-low)/numSteps;
    double area = 0.0;
    while( low < high ) {
        area += f( low ) * step;
        low += step;
    }
    return area;
}
```

In this version, we pass a pointer to the function over which we want to integrate.

```cpp
double aFunc( double x ) { ... } //...
double area = integrate( aFunc, 0.0, 2.71828 );
```

This works, but it's inflexible because it uses a function pointer to indicate the function to be integrated; it can't handle functions that require state or pointers to member functions. An alternative is to create a function object hierarchy. The base of the hierarchy is a simple interface class that declares a pure virtual operator ().

```cpp
class Func {
    public:
        virtual ~Func();
        virtual double operator ()( double ) = 0;
    }
    double integrate( Func &f, double low, double high );
```
Now *integrate* is capable of integrating any type of function object that is a *Func* (see *Polymorphism* [2, 3]). It's also interesting to note that the body of *integrate* does not have to change at all (though it does require recompilation), because we use the same syntax to call a function object as we do for a pointer to function. For example, we can derive a type of *Func* that can handle non-member functions:

```cpp
class NMFunc : public Func {
    public:
        NMFunc( double (*f)( double ) ) : f_(f) {}
        double operator ()( double d ) { return f_(d); }
    private:
        double (*f_)( double );
};
```

This allows us to integrate all the functions of our original version:

```cpp
double aFunc( double x ) { ... }
//...
NMFunc g( aFunc );
double area = integrate( g, 0.0, 2.71828 );
```

We can also integrate member functions by wrapping an appropriate interface around a pointer to member function and a class object (see *Pointers to Member Functions Are Not Pointers* [16, 57]):

```cpp	template <class C>
class MFunc : public Func {
    public:
        MFunc( C &obj, double (C::*f)(double) )
            : obj_(obj), f_(f) {}
        double operator ()( double d )
            { return (obj_::*f_)(d); }
    private:
        C &obj_;
        double (C::*f_)( double );
};
//...
AClass anObj;
MFunc<AClass> f( anObj, &AClass::aFunc );
double area = integrate( f, 0.0, 2.71828 );
```
Item 20. STL Function Objects

How did we ever get by without the STL? Not only is it easier and faster to write complex code, but that code is both standard and highly optimized.

```cpp
std::vector<std::string> names;
//...
std::sort( names.begin(), names.end() );
```

Another nice thing about the STL is that it's highly configurable. In the code above, we used string's less-than operator to sort a vector of strings, but we don't always have a less-than operator to work with, or we may not want to sort in ascending order.

```cpp
class State {
  public:
    //...
    int population() const;
    float aveTempF() const;
    //...
};
```

The State class (which represents a state of the union) doesn't have a less-than operator, and we probably don't want to implement one because it's not clear what it would mean for one state to be less than another (do we compare names, population, percentage of elected officials under indictment, ...?). Fortunately, the STL generally allows us to specify an alternate less-than-like operation in situations like this. Such an operation is called a "comparator," because it compares two values:

```cpp
inline bool popLess( const State &a, const State &b )
{ return a.population() < b.population(); }
```

Once we have a comparator for States, we can use it to sort them:

```cpp
State union[50];
//...
std::sort( union, union+50, popLess ); // sort by population
```

Here we've passed a pointer to the popLess function as the comparator (recall that a function name "decays" into a pointer to function when passed as an argument, just as the array name union decays into a pointer to its first element). Because popLess is passed as a function pointer, it will not be inlined in sort, which is unfortunate if we want a fast sort operation (see Function Pointers [14, 49]).
We can do better if we use a function object as a comparator:

```cpp
struct PopLess : public std::binary_function<State, State, bool> {
    bool operator()(const State &a, const State &b) const
    {
        return popLess(a, b);
    }
};
```

The `PopLess` type is a typical example of a properly constructed STL function object. First, it's a function object. It overloads the function call operator so that it may be called with the usual function call syntax. This is important, because STL generic algorithms like `sort` are written in such a way that either a function pointer or function object may be used to instantiate them, provided that they may be called with the typical function call syntax; a function object with an overloaded `operator()` satisfies this syntactic requirement.

Second, it's derived from the standard `binary_function` base class. This is a mechanism that allows other parts of the STL implementation to ask compile-time questions of the function object (see `Embedded Type Information` [53, 189]). In this case, deriving from `binary_function` allows one to find out the argument and return types of the function object. We're not using that capability here, but you can bet that somebody else will, and we want our `PopLess` type to be used by others.

Third, the function object has no data members, no virtual functions, and no explicitly declared constructors or destructor, and the implementation of `operator()` is inline. Function objects used as STL comparators are assumed to be small, simple, and fast. It's possible to design STL function objects with significant implementations, but it's rarely advisable. Another reason to avoid or minimize the use of data members in a function object to be used with the STL is that STL implementations may make several copies of a function object and may assume that all the copies are identical. One easy way to ensure that all copies of an object are identical is for the object to have no data at all.

Now we can sort this country out by using a function object:

```cpp
sort(union, union+50, PopLess());
```

Note the parentheses that follow `PopLess` in this call to `sort`. `PopLess` is a type, but we have to pass an object of that type as a function argument. By appending parentheses to the `PopLess` type name, we create an unnamed temporary `PopLess` object that exists for the duration of the function call. (These nameless objects are known as "anonymous temporaries," a term I've always enjoyed because it sounds vaguely racy.) We could have declared and passed a named object:

```cpp
PopLess comp;
sort(union, union+50, comp);
```
However, it's more conventional, and less typing, simply to pass an anonymous temporary object.

A beneficial side effect of using a function object as our comparator is that the comparison will be inlined whereas use of a function pointer did not permit inlining. The reason the call is inlined is that the compiler knows that the type of the comparator is `PopLess` when the `sort` function template is instantiated, which in turn allows it to know that `PopLess::operator()` will be called, which in turn allows it to inline that function, which in turn allows it to inline the nested call to `popLess`.

Another common use of a function object in the STL is as a predicate. A predicate is an operation that asks a true/false question about a single object. (You can think of a comparator as a kind of binary predicate.)

```cpp
struct IsWarm : public std::unary_function<State,bool> {
    bool operator ()( const State &a ) const
    { return a.aveTempF() > 60; }
};
```

The design guidelines for STL predicates are identical to those for STL comparators with the exception, of course, that they're unary rather than binary functions. Starting with our previous sorted `State` results, the appropriate predicate makes it trivial to find a warm place without too many dang people:

```cpp
State *warmandsparse = find_if( union, union+50, IsWarm() );
```