# Functors: Function Objects in C++

By Alex Allain

Both C and C++ support function pointers, which provide a way to pass around instructions on how to perform an operation. But function pointers are limited because functions must be fully specified at compile time. What do I mean? Let's say that you're writing a mail program to view an inbox, and you'd like to give the user the ability to sort the inbox on different fields--to, from, date, etc. You might try using a sort routine that takes a function pointer capable of comparing the messages, but there's one problem--there are a lot of different ways you might want to compare messages. You could create different functions that differ only by the field of the message on which the comparison occurs, but that limits you to sorting on the fields that have been hard-coded into the program. It's also going to lead to a lot of if-then-else blocks that differ only by the function passed into the sort routine.

What you'd really like is the ability to pass in a third argument to your comparison function, telling it which field to look at. But to make this work, you'd have to write your own sort routine that knows about the third argument; you can't use a generic routine like the STL's sort function because you can't tell it to pass in a third argument to the comparator. Instead, you somehow need the ability to "embed" what field to sort on inside the function.

It turns out that you can get this behavior in C++ (but not in C) through the use of function objects (aka "functors"). Functors are objects that can be treated as though they are a function or function pointer--you could write code that looks like this:

```
1  myFunctorClass functor;
2  functor( 1, 2, 3 );
```

This code works because C++ allows you to overload operator(), the "function call" operator. The function call operator can take *any* number of arguments of *any* types and return anything it wishes to. (It's probably the most flexible operator that can be overloaded; all the other operators have a fixed number of arguments.) For ease of discussion, when an object's operator() is invoked, I'll refer to it as "calling" the object as though it were a function being called.

While overloading operator() is nice, the really cool thing about functors is that their lifecycle is more flexible than that of a function--you can use a a functor's constructor to embed information that is later used inside the implementation of operator().

Let's look at an example. This example creates a functor class with a constructor that takes an integer argument and saves it. When objects of the class are "called", it will return the result of adding the saved value and the argument to the functor:

```
1  #include <iostream>
2
3  class myFunctorClass
4  {
5      public:
6          myFunctorClass (int x) : _x( x ) {}
7          int operator() (int y) { return _x + y; }
8      private:
9          int _x;
10 };
11
```

```
int main()
{
    myFunctorClass addFive( 5 );
    std::cout << addFive( 6 );

    return 0;
}
```

In short, the act of constructing an object lets you give the functor information that it can use inside the implementation of its function-like behavior (when the functor is called through operator()).

## Sorting Mail

Now we can think about how we'd want to write a mail sorting functor. We'd need operator() to take two messages, and we can have its constructor store the field we wish to sort by.

```
class Message
{
    public:
        std::string getHeader (const std::string& header_name) const;
        // other methods...
};

class MessageSorter
{
    public:
        // take the field to sort by in the constructor
        MessageSorter (const std::string& field) : _field( field ) {}
        bool operator() (const Message& lhs, const Message& rhs)
        {
            // get the field to sort by and make the comparison
            return lhs.getHeader( _field ) < rhs.getHeader( _field );
        }
    private:
        std::string _field;
};
```

Now if we had a vector of Messages, we could use the STL sort function to sort them:

```
std::vector<Message> messages;
// read in messages
MessageSorter comparator;
sort( messages.begin(), messages.end(), comparator );
```

## Functors Compared with Function Pointers

If you have a function that takes a function pointer, you cannot pass in a functor as though it were a function pointer, even if the functor has the same arguments and return value as the function pointer.

Likewise, if you have a function that expects a functor, you cannot pass in a function pointer.

## Functors, Function Pointers, and Templates

If you wish to allow either a function pointer or a functor to be passed into the same function, you need to use templates. The templated function will deduce the proper type for the functor or function pointer, and both functors and function pointers are used in the exact same way--they both look like function calls--so the code in the function will compile fine.

For instance, let's say you had this simple little find_matching_numbers function that takes a vector and returns a vector of all numbers for which a given function returns true.

```
std::vector<int> find_matching_numbers (std::vector<int> vec, bool (*pred)(int))
{
    std::vector<int> ret_vec;
    std::vector<int>::iterator itr = vec.begin(), end = vec.end();
    while ( itr != end )
    {
        if ( pred( *itr ) )
        {
            ret_vec.push_back( *itr );
        }
        ++itr;
    }
}
```

We could quickly convert this function to allow either functors or function pointers by templating on the type of the second argument:

```
template <typename FuncType>
std::vector<int> find_matching_numbers( std::vector<int> vec, FuncType pred )
{
    std::vector<int> ret_vec;
    std::vector<int>::iterator itr = vec.begin(), end = vec.end();
    while ( itr != end )
    {
        if ( pred( *itr ) )
        {
            ret_vec.push_back( *itr );
        }
        ++itr;
    }
}
```

Notice that this also lets you avoid having to use the rather convoluted function pointer syntax. Sweet!

## Functors vs. Virtual Functions

Functors and virtual functions are, believe it or not, closely related. They both solve the same problem: how do we let some code choose the algorith that is applied. Imagine that you have a function called doMath, and it takes 3 arguments: two integers, and a "kind of math to do". You could write it with functors like this:

```
template <FuncType>
int doMath (int x, int y, FuncType func)
{
    return func( x, y );
}
```

Another way that you could write this is by creating an interface class that has a computeResult method:

```
class MathComputer
{
    virtual int computeResult (int x, int y) = 0;
};

doMath (int x, int y, MathComputer* p_computer)
{
    return p_computer->computeResult( x, y );
}
```

Admittedly, this is kind of a boring example! All it shows you is that both virtual functions and functors allow you to dynamically choose the exact algorithm that is executed.

The major difference is that using virtual functions does not give the ability to write a templated function that can also accept function pointers. Depending on your application, this may be important: if you're writing a library, it's probably very important; if you're writing a small program for yourself, it's likely less important. A virtual function, on the other hand, has somewhat simpler syntax (no complex templates!) and tends to fit the normal object-oriented programming mindset.

**Related Articles**

Lambda functions in C++11

Function Pointers in C and C++

C++ Operator Overloading