Functors

Introduction

Functors (also called *function objects* or *functionoids*) are a strange but incredibly useful feature of the C++ language that are essentially "smart functions." While initially functors can be a bit confusing, with practice you will come to appreciate their immense power, flexibility, and versatility. This handout serves as an introduction to functors in preparation for Handout #21 on functional programming with the STL.

A Simple Problem

To understand the motivation behind functors, let's consider a simple task and how we might try to solve it using the STL algorithms. Let's suppose you have a `vector<string>` containing random strings and you'd like to count the number of elements in the `vector` that have length less than five. You stumble upon the `count_if` STL algorithm, which accepts a range of iterators and a predicate function and returns the number of elements in the iterator range for which the function returns true. Since we want to count the number of strings with length less than five, we could then write a function like this one:

```
bool LengthIsLessThanFive(const string &str)
{
    return str.length() < 5;
}
```

And then call `count_if(myVector.begin(), myVector.end(), LengthIsLessThanFive)` to get the number of short elements. Similarly, if you wanted to count the number of strings with length less than ten, you could write a `LengthIsLessThanTen` function, and so on and so forth. While this approach will work in some cases, it is critically hampered because the maximum length must be determined at compile-time. For example, suppose you want to write a program that prompts the user for a number and then returns the number of strings in the `vector` with fewer than that many characters. Since the user could enter any integer value, you cannot use the above approach, since we couldn't determine at compile-time which value to compare the string's length against.

To solve this problem, you might consider writing a function like this:

```
bool LengthIsLessThan(const string &str, int length)
{
    return str.length() < length;
}
```

While this is indeed far more generic than the above approach, it won't work in conjunction with `count_if` since `count_if` requires a unary function (a function taking only one argument) as a parameter. Somehow we need to build a unary function that stores the value to compare against the string length. While we can do this with global variables, that approach is horrendously unsightly and can lead to all sorts of other problems (especially if two different functions each need to modify the global variable). Instead, we'll use a *functor*.

At a high level, a functor is an object that acts like a function. For example, suppose we create a functor class called `MyClass` that imitates a function accepting an `int` parameter and returning a `double`. Then the following code would be legal:

```
MyClass myFunctor;
cout << myFunctor(137) << endl; // "Call" myFunctor with parameter 137
```

Just like a regular function, here we can "call" the `myFunctor` object with whatever parameter we want.

To create a functor, you create an object that overloads the parentheses operator, `operator ()`. Unlike other operators we've seen so far, when overloading the parentheses operator, you're free to return an object of any type (or even `void`) and can accept any number of parameters. For example, here's a sample functor that overloads the parentheses operator to print out a string:

```
struct MyFunctor
{
    void operator() (const string &str) const
    {
        cout << str << endl;
    }
};
```

Note that there are two sets of parentheses there. The first group is for the function name – `operator ()` – and the second for the parameters to `operator ()`. Also, note that we're using a `struct` instead of a `class`. C++ makes no distinction between `struct`s and `class`es other than the default visibility, and to conserve space most C++ programmers make their functors `struct`s instead of `class`es. To use this functor, we can write:

```
MyFunctor functor;
functor("Functor power!");
```

Which prints out "Functor power!"

In the above example, our functor acted just like a regular function (and indeed, we could have completely replaced the functor with a C++ function). However, functors are immensely powerful because unlike regular C++ functions, they can store and retrieve information beyond that provided by their parameters. For example, consider the following functor class:

```
struct StringAppender
{
    /* Constructor takes and stores a string. */
    explicit StringAppender(const string &str) : toAppend(str) {}

    /* operator() prints out a string, plus the stored suffix. */
    bool operator() (const string &str) const
    {
        cout << str << ' ' << toAppend << endl;
    }
    const string toAppend;
};
```

This object represents a functor whose constructor takes in a string and whose `operator ()` function prints out a string parameter, followed by the stored string. We use the `StringAppender` functor like

this:

```
StringAppender myFunctor("is awesome");
myFunctor("C++");
```

This code will print out "C++ is awesome," since we passed in "C++" as a parameter and the functor appended its stored string "is awesome." Basically, we've written something that looks like a single-parameter function but that has access to extra information. This is the critical difference between a function and a functor – while a function cannot access any information beyond its parameters, a functor has access to both its parameters and all of its data members.

Let's return to the above example with `count_if`. Somehow we need to provide a unary function that can return whether a string is less than an arbitrary length. To solve this problem, instead of writing a unary function, we'll create a unary *functor* whose constructor stores the maximum length and whose `operator ()` accepts a string and returns whether it's of the correct length. Here's one possible implementation:

```
struct ShorterThan
{
    /* Accept and store an int parameter */
    explicit ShorterThan(int maxLength) : length(maxLength) {}

    /* Return whether the string length is less than the stored int. */
    bool operator() (const string &str) const
    {
        return str.length() < length;
    }
    const int length;
};
```

Note that while `operator ()` takes in only a single parameter, it has access to the `length` field that was set up by the constructor. This is exactly what we want – a unary function that somehow knows what value to compare the parameter to. To tie everything together, here's the code we'd use to count the number of strings in the `vector` that are shorter than the specified value:

```
ShorterThan st(length);
count_if(myVector.begin(), myVector.end(), st);
```

Functors are absolutely incredible when combined with STL algorithms for this very reason – they look and act like regular functions but have access to extra information. This is just your first taste of functors, and there are some absolutely incredible things you can do with functors that have significant implications for the way you program using the STL, as detailed in the next handout.

Creating Temporary Objects

When working with functors, you'll commonly want to create a temporary class instance that exists only in the context of a function call. While right now you might be a bit confused about exactly why you'd ever want to do this, it should become clearer shortly.

In C++, you are allowed to create temporary objects for the duration of a single line of code by explicitly calling the object's constructor. For example, the following code creates a temporary `vector<int>` and prints out its size:

```
cout << vector<int>().size() << endl;
```

Let's analyze exactly what's going on here. The code `vector<int>()` creates a temporary `vector<int>` object by calling the `vector<int>` constructor with no parameters. Therefore, the newly-created `vector` has no elements. We then call the temporary `vector<int>`'s `size` member function, which will return zero since the `vector` is empty. Once this line finishes executing, the `vector`'s destructor will invoke, cleaning up the new object.

While the above example is admittedly quite useless, it is important to know that it's legal to construct objects "on the fly" using this syntax because it frequently arises in professional code. Look back to the above code with `count_if`. If you'll notice, we're creating a new `ShorterThan` class using the parameter `length`, then feeding the object to `count_if`. After that line, odds are that we'll never use the `ShorterThan` object again. This is an excellent spot to use temporary objects, since we need a new `ShorterThan` for the function call but don't plan on using it afterwards. Thus, we can convert this code:

```
ShorterThan st(length)
count_if(myVector.begin(), myVector.end(), st);
```

Into this code:

```
count_if(myVector.begin(), myVector.end(), ShorterThan(length));
```

Here `ShorterThan(length)` constructs a temporary `ShorterThan` functor with parameter `length`, then passes it to the `count_if` algorithm. Don't get tripped up by the syntax – `ShorterThan(length)` does *not* call the `ShorterThan`'s `operator ()` function. Instead, it invokes the `ShorterThan` constructor with the parameter `length` to create a temporary object.

Storing Objects in STL `maps`, Part II

In the previous handout, we demonstrated how to store custom objects as keys in an STL `map` by overloading the < operator. However, what if you want to store elements in a `map` or `set`, but not using the default comparison operator? For example, consider a `set<char *>` of C strings. Normally, the < operator will compare two `char *`s by seeing if one references memory with a lower address than the other. This isn't at all the behavior we want. First, it would mean that the `set` would store its elements in a seemingly random order since the comparison is independent of the contents of the C strings. Second, if we tried to call `find` or `count` to determine membership in the `set`, since the `set` compares the *pointers* to the C strings, not the C strings themselves, `find` and `count` would return whether the given pointer, not the pointee, was contained in the `set`.

We need to tell the `set` that it should not use the < operator to compare C strings, but we can't simply provide an alternative < operator and expect the `set` to use it. Instead, we'll define a functor class whose `operator ()` compares two C strings lexicographically and returns whether one string compares less than the other. Here's one possible implementation:

```
struct CStringCompare
{
    bool operator() (const char *one, const char *two) const
    {
        return strcmp(one, two) < 0; // Use strcmp to do the comparison
    }
};
```

Then, to signal to the `set` that it should store elements using `CStringCompare` instead of the default `<` operator, we'll define the `set` as a `set<char *, CStringCompare>`. Note that we specify the comparison functor class as a template argument to the `set`. This means that `set<char *>` and `set<char *, CStringCompare>` are two different types, so you can only iterate over a `set<char *, CStringCompare>` with a `set<char *, CStringCompare>::iterator`. `typedef` will be your ally here. You can use a similar trick for the `map` by declaring a `map<KeyType, ElemType, CompareType>`.

Writing Functor-Compatible Code

In CS106X, you've seen how to write code that accepts a function pointer as a parameter. For example, consider the following code, which accepts a function that takes and returns a `double`, then prints a table of some sample values of the function:

```
void TabulateFunctionValues(double (function)(double))
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Assume that we have some unary functor `MyFunctor` that accepts and returns a `double` and that we want to pass this functor into `TabulateFunctionValues`. Unfortunately, as it is currently written, `TabulateFunctionValues` cannot accept a `MyFunctor` object as a parameter, since its parameter is defined as `double (function)(double)` and not `MyFunctor function`. To solve this problem, redefine `TabulateFunctionValues` as a template function that accepts as a parameter an object of some template type `UnaryFunction`, as shown here:

```
template<typename UnaryFunction>
void TabulateFunctionValues(UnaryFunction function)
{
    for(double i = LOWER_BOUND; i <= UPPER_BOUND; i += STEP)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Now, we can pass both functions and function pointers into `TabulateFunctionValues`, since for any type we pass in we will get a newly-created template instantiation that accepts a parameter of that type.

If you'll notice, the above code is templatized over some type called `UnaryFunction`. Like any other template function or template class, this means that we are allowed to pass an object of any type we'd like to `TabulateFunctionValues`. Does this pose a problem if we try to pass in an `int` or `string` that isn't a functor or function? The answer is no. Although `UnaryFunction` can be an object of any type, if we provide as an argument a value that cannot be called as a function, we will get a compile-time error. In C++ jargon, this is known as an *implicit interface*. There are no formal restrictions on what sorts of objects we can provide as arguments to `TabulateFunctionValues`, but only those types that can be called as a unary function accepting a `double` will result in legal code. Implicit interfaces are an advanced topic in C++, so if you're interested, consult a reference for more information.

Practice Problems

1. All overloaded operators except for `operator ()` only allow you to specify a fixed number of parameters. What about `operator ()` makes this rule not hold?
2. The CS106X `Grid` class allows you to access individual elements using the syntax `myGrid(x, y)`. How is this functionality implemented? Given what you know about functors, do you consider this an appropriate use of the parentheses operator?
3. The STL algorithm `for_each` accepts as parameters a range of iterators and a unary function, then calls the function on each argument. Unusually, the return value of `for_each` is the unary function passed in as a parameter. Why might this be?
4. Using the fact that `for_each` returns the unary function passed as a parameter, write a function `MyAccumulate` that accepts as parameters a range of `vector<int>::iterator`s and an initial value, then returns the sum of all of the values in the range, starting at the specified value. Do not use any loops – instead, use `for_each` and a custom functor class that performs the addition.
5. Write a function `AdvancedBiasedSort` that accepts as parameters a `vector<string>` by reference and a `string` "winner" parameter, then sorts the `vector`, except that all strings equal to the winner string are at the front of the `vector`. Do not use any loops. *(Hint: Use the STL `sort` algorithm and functor that stores the "winner" parameter.)*
6. The STL `generate_n` algorithm is defined as `void generate_n(OutputIterator start, size_t count, NullaryFunction fn)` and calls the zero-parameter function `fn` `count` times, storing the output in the range beginning at `start`. Write a function `FillAscending` that accepts two parameters, an empty `vector<int>` by reference and an `int` called `n` and fills the `vector` with the integers in the range [`0`, `n`). Do not use any loops.
7. Write a function `VectorToMap` that accepts a `vector<string>` and an integer value and returns a `map<string, int>` whose keys are equal to the strings in the vector and whose values are equal to the `int` parameter. Do not use any loops. *(Hint: Use `transform` and a callback functor. Remember that `map`s store elements as `pair<const KeyType, ValueType>`s)*