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 #28 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.

The above approach is perfectly legal C++, but is not particularly elegant. Rather than writing multiple different functions to compare string lengths, intuitively it seems like we should just write one function that looks like this:

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

This way, we can specify what the maximum length is in the second parameter.

The trouble with this function is that we can't use it in conjunction with `count_if` to solve the above problem because `count_if` requires a unary function (a function taking only one argument) as its final parameter. The `LengthIsLessThan` function is a binary function, even though we plan on always passing a fixed value as the second parameter.

What we need is a way to construct a function that takes in only one parameter (the string to test), but which has access to the maximum length for the string. Using an object called a *functor*, we can resolve this problem.

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 CS106B/X, 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.

STL Algorithms Revisited

Now that you're armed with the full power of C++ functors, let's revisit some of the STL algorithms we covered earlier in the quarter and discuss how to maximize their firepower.

The very first algorithm we covered in CS106L was `accumulate`, defined in the `<numeric>` header. If you'll recall, `accumulate` sums up the elements in a range and returns the result. For example, given a `vector<int>`, the following code returns the sum of all of the `vector`'s elements:

```
accumulate(myVector.begin(), myVector.end(), 0);
```

The first two parameters should be self-explanatory, and the third parameter (zero) represents the initial value of the sum.

However, this view of `accumulate` is surprisingly limited, and to treat `accumulate` as simply a way to sum container elements would be an error. Rather, *`accumulate` is a general-purpose function for transforming a collection of elements into a single value.*

There is a second version of the `accumulate` algorithm that takes a binary function as a fourth parameter. This version of `accumulate` is implemented like this:

```
template<typename InputIterator, typename Type, typename BinaryFn> inline
Type accumulate(InputIterator start, InputIterator stop, Type initial,
                BinaryFn fn)
{
   while(start != stop)
   {
      initial = fn(initial, *start);
      ++start;
   }
   return initial;
}
```

This `accumulate` iterates over the elements of a container, calling the provided binary function on the current value of the variable `initial` (called the *accumulator*) and the next element of the container, then storing the result back in `initial`. In other words, `accumulate` continuously updates the value of the accumulator based on its initial value and the values contained in the input range. Finally, `accumulate` returns the value stored in `initial`. Note that the version of `accumulate` we encountered earlier in the quarter is actually a special case of the above version where the provided callback function computes the sum of its parameters.

To see an example of `accumulate` in action, let's consider an example, Recall that the STL algorithm `lower_bound` returns an iterator to the first element in a range that compares greater than or equal to some value. However, `lower_bound` requires the elements in the iterator range to be in sorted order, so if you have an unsorted `vector`, you cannot use `lower_bound`. Let's write a function `UnsortedLowerBound` that accepts a range of iterators and a lower bound, then returns the value of the lowest element in the range greater than or equal to the lower bound. For simplicity, let's assume we're working with a `vector<int>` so that we don't get bogged down in template syntax, though this approach can easily be generalized.

Although this function can be implemented using loops, we can leverage off of `accumulate` to come up with a considerably more concise solution. Thus, we'll define a functor class to pass to `accumulate`, then write `UnsortedLowerBound` as a wrapper call to `accumulate` with the proper parameters. Consider the following functor:

```
struct LowerBoundHelper
{
    const int lowestValue;
    explicit LowerBoundHelper(int lower) : lowestValue(lower) {}
    int operator() (int bestSoFar, int current)
    {
        if(current >= lowestValue && current < bestSoFar) return current;
        return bestSoFar;
    }
};
```

This functor's constructor accepts the value that we want to lower-bound. Its `operator ()` function accepts two `int`s, the first representing the current lowest value and the second the lowest value greater than `lowestValue` we've encountered so far. If the value of the current element is greater than or equal to the lower bound and also less than the best value so far, `operator ()` returns the value of the current element. Otherwise, it simply returns the best value we've found so far. Thus if we call this functor on every element in the `vector` and keep track of the return value, we should end up with the lowest value in the `vector` greater than or equal to the lower bound. We can now write the `UnsortedLowerBound` function like this:

```
int UnsortedLowerBound(const vector<int> &input, int lowerBound)
{
    return accumulate(input.begin(), input.end(), INT_MAX,
                      LowerBoundHelper(lowerBound));
}
```

Our entire function is simply a wrapped call to `accumulate`, passing a specially-constructed `LowerBoundHelper` object as a parameter. Note that we've specified the starting parameter as `INT_MAX`, a constant defined in the `<climits>` header file that contains the largest possible value of an `int`. That way, if there is a value in the container greater than the lower bound, on some iteration of the `LowerBoundHelper` call, that value will replace `INT_MAX` as the maximum value. Otherwise, if none of the elements are greater than the lower bound, the function will return `INT_MAX` as a sentinel.

If you need to transform a range of values into a single result (of any type you wish), use `accumulate`. To transform a range of values into another range of values, use `transform`. We discussed `transform` briefly earlier in the quarter in the context of `ConvertToUpperCase` and `ConvertToLowerCase`, but such examples are just the tip of the iceberg. `transform` is nothing short of a miracle function, and it arises a whole host of circumstances.*

* Those of you familiar with functional programming might recognize `accumulate` and `transform` as the classic higher-order functions Map and Reduce.

Practice Problems

1. 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?
2. 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.
3. 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.)*
4. 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.
5. 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)*