CS106L
Winter 2007-2008

Handout #22
March 5, 2008

The STL `<functional>` Library

Introduction

Up to this point, all of the programs you have encountered in CS106 have used immutable functions to create and modify mutable data. This approach is referred to as the *imperative programming paradigm* and inherits from a tradition rooted in C and assembly language. However, there is another scheme for writing programs, the *functional programming paradigm*, where functions themselves can be created and modified.

Functional programming is an entirely different way of approaching a programming problem and to discuss it in detail is far beyond the scope of this class. However, using a restricted subset of functional programming through the STL functional libraries, we can supercharge C++ code, especially in conjunction with STL algorithms. This handout discusses how to increase clarity and reduce code reuse by leveraging off of the STL functional library.

Adaptable Functions

To provide functional programming support, standard C++ provides the `<functional>` library. `<functional>` exports several useful functions that can transform and modify functions on-the-fly to yield new functions more suitable to the task at hand. However, because of C++'s natural limitations, the `<functional>` library can only modify specially constructed functions called "adaptable functions," funct*ors* (not regular C++ functions) that inherit from one of two template base classes, `unary_function` and `binary_function`. In case you haven't seen inheritance yet, don't worry – all you'll need to do is add an extra line to your functor declarations.

To construct an adaptable function, you first construct a `struct` overloading `operator ()` and then have it inherit from the proper `unary_function` or `binary_function` class. For example, let's suppose you want to make an adaptable function called `MyFunction` that takes a `string` by reference-to-`const` as a parameter and returns a `bool`. You start by writing `MyFunction` as a functor that looks like this:

```
struct MyFunction
{
    bool operator() (const string &str) const
    {
        /* Function that manipulates a string */
    }
};
```

To make this functor an adaptable function, we'll make it inherit from `unary_function`. Because `unary_function` is a template class, we must explicitly specify its template arguments. `unary_function` is defined as a two-argument template class that looks like this:

```
template<typename ParameterType, typename ReturnType>
   class unary_function;
```

The first template argument represents the type of the parameter to the function; the second, the function's return type. Thus, to make `MyFunction` an adaptable function, we'll need to specify that it inherits from `unary_function<string, bool>`, as shown here:

```
struct MyFunction : public unary_function<string, bool> // Correct
{
    bool operator() (const string &str) const
    {
        /* Function that manipulates a string */
    }
};
```

In case you haven't seen inheritance yet, the public `unary_function<string, bool>` simply tells C++ to import all of the data from the class `unary_function<string, bool>` into this class.

Note that although the function accepts as its parameter a `const string &`, in the template declaration we nonetheless use a regular `string`. The reason is somewhat technical and has to do with how `unary_function` interacts with other functional library components, so for now just remember that you should not specify reference-to-`const` types inside the `unary_function` template parametrization. When writing code using adaptable functions, deriving your functor from the wrong specialization of `unary_function` can lead to absolutely hideous compiler errors, so make sure that you have the parameter order correct.

Now suppose that we'd like to make an adaptable binary function that accepts a `string` and an `int` and returns a `bool`. We begin by writing the basic functor code, as shown here:

```
struct MyOtherFunction
{
    bool operator() (const string &str, int val) const
    {
        /* Do something, return a bool. */
    }
};
```

To convert this functor into an adaptable function, we'll have it inherit from `binary_function`. Like `unary_function`, `binary_function` is a template base class that's defined as

```
template<typename Param1Type, typename Param2Type, typename ResultType>
    class binary_function;
```

Thus the adaptable version of `MyOtherFunction` would be

```
struct MyOtherFunction: public binary_function<string, int, bool>
{
    bool operator() (const string &str, int val) const
    {
        /* Do something, return a bool. */
    }
};
```

ptr_fun

While the above approach for generating adaptable functions is perfectly legal, it's a bit clunky and we have a high ratio of boilerplate code to actual logic. Fortunately, the STL functional library provides the powerful but cryptically named ptr_fun* function that transforms a regular C++ function into an adaptable function. ptr_fun can converts both unary and binary C++ functions into adaptable functions with the correct parameter types, meaning that you can skip the hassle of the above code by simply writing normal functions and then using ptr_fun to transform them into adaptable functions. You'll see some examples of ptr_fun later in this handout.

Unfortunately, you cannot use ptr_fun on functions that accept parameters as reference-to-const. ptr_fun returns a unary_function object, and as mentioned above, you cannot specify reference-to-const as template arguments to unary_function. If you want the speed boost of pass-by-reference when working with adaptable functions, you'll need to use the full functor syntax.

Binding Parameters

Now that we've covered how the STL functional library handles adaptable functions, let's consider how we can use them in practice. Last week, we discussed functors and how to use them to create unary functions that have access to extra state information. Let's return to the example with count_if, where we wanted to use STL algorithms to count the number of strings in a container whose length was less than a certain threshold. Our solution was to construct a functor called ShorterThan whose constructor accepted as a parameter the cutoff length and whose operator () function returned whether a string's length was less than the specified value. Let's take a look at that code again:

```
struct ShorterThan
{
    explicit ShorterThan(int maxLength) : length(maxLength) {}
    bool operator() (const string &str) const
    {
        return str.length() < length;
    }
private:
    int length;
};
```

The functor's operator () function is a trivial one-line function, but if you'll notice, the ShorterThan class definition is ten lines long even with some space-saving tricks. There is almost three times as much boilerplate code as there is actual logic. This isn't a problem *per se*, but it is certainly an inconvenience. Let's take a step back from functors and see if we can find another approach.

The fundamental problem is that the STL count_if algorithm requires a single-parameter function, but the function we want to use as a callback is a binary function. We want the STL algorithms to use our two-parameter function LengthIsLessThan, but with the second parameter always having the same value. What if somehow we could modify LengthIsLessThan by "locking in" the second parameter? In other words, we'd like to take a function that looks like this:

* ptr_fun is short for "pointer function", since you're providing as a parameter a function pointer. It should not be confused with "fun with pointers."

```
bool LengthIsLessThan

string str
(parameter)

int length
(parameter)
```

And transform it into another function that looks like this:

```
bool LengthIsLessThan

string str
(parameter)

int length
5
```

This approach, called *parameter binding*, is an excellent example of functional programming supported by the STL functional library. The general idea is to take a function that takes N parameters, then bind one of them to a certain value so that we end up with an N-1 parameter function.

To bind a parameter to a function, we can use the `bind2nd` function, a two-parameter function that accepts an adaptable function and the value to bind and returns a new adaptable function equal to the original function but with the second parameter locked in place. So, given the following implementation of `LengthIsLessThan`:

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

We could use the following syntax to construct a function that's `LengthIsLessThan` with the value five bound to the second parameter:

```
bind2nd(ptr_fun(LengthIsLessThan), 5);
```

The line `bind2nd(ptr_fun(LengthIsLessThan), 5)` first uses `ptr_fun` to generate an adaptable version of the `LengthIsLessThan` function, then uses `bind2nd` to lock the parameter five in place. The result is a new unary function that accepts a `string` and returns whether its length is strictly less than five. Thus, we could count the number of `string`s in a container with length less than five by writing

```
count_if(container.begin(), container.end(),
         bind2nd(ptr_fun(LengthIsLessThan), 5));
```

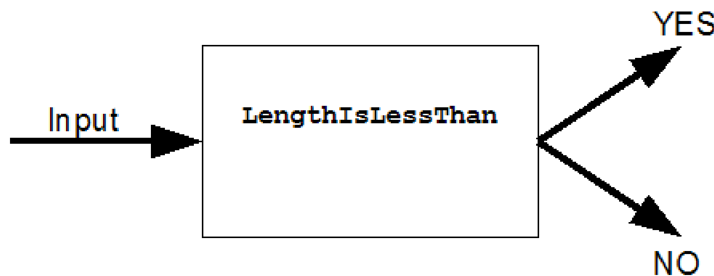
In the STL functional library, parameter binding is restricted only to binary functions. Thus you cannot bind a parameter in a three-parameter function to yield a new binary function, nor can you bind the parameter of a unary function to yield a zero-parameter ("nullary") function. For these operations, you'll need to use functors, as shown last week.*

Just as there is a `bind2nd` function that binds the second parameter of a function, there is also a `bind1st` function that binds the first parameter of a function. Returning to the above example, given a `vector<int>`, we could count the number of elements in that `vector` smaller than the length of string "C++!" by writing

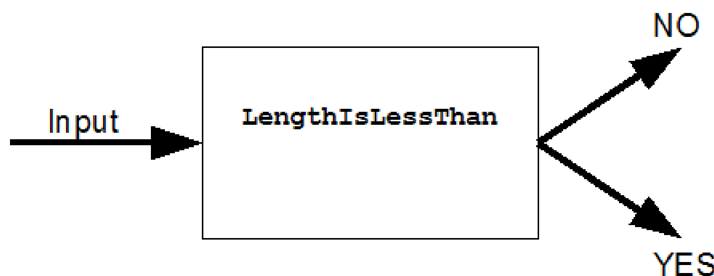
```
count_if(myVector.begin(), myVector.end(),
         bind1st(ptr_fun(LengthIsLessThan), "C++!"));
```

Negating Results

Let's consider another functional programming problem we might encounter. Suppose that given a function `LengthIsLessThan`, we want to find the number of strings in a container that are *not* less than a certain length. While we could simply write another function `LengthIsNotLessThan`, it would be much more convenient if we could somehow tell C++ to take whatever value `LengthIsLessThan` returns and to use the opposite result. That is, given a function that looks like this:



We'd like to change it into a function that looks like this:



This operation is called *negation* – constructing a new function whose return value has the opposite value of the input function. There are two STL negator functions – `not1` and `not2` – that return the negated result of a unary or binary predicate function, respectively. Thus, the above function that's a negation of `LengthIsLessThan` could be written as `not2(ptr_fun(LengthIsLessThan))`. Since `not2` returns an adaptable function, we can then pass the result of this function to `bind2nd` to generate a unary function that returns whether a string's length is at least a certain threshold value. For example, here's

---

* It is possible to create functions and functor classes that can bind parameters of arbitrary functions, and if you're up for a challenge, it might be worth it to see how functional programming constructs work behind the scenes.

code that returns the number of `string`s in a container with length at least 5:

```
count_if(container.begin(), container.end(),
         bind2nd(not2(ptr_fun(LengthIsLessThan)), 5));
```

While this line is incredibly dense, it quite elegantly solves the problem at hand by combining and modifying existing code to create entirely different functions. Such is the beauty and simplicity of functional programming – why rewrite code from scratch when you already have all the pieces individually assembled?

Operator Functions

Let's suppose that you have a container of `int`s and you'd like to add 137 to each of them. Recall that you can use the STL `transform` algorithm to apply a function to each element in a container and then store the result. Since you're adding 137 to each element, you might consider writing a function like this one:

```
int Add137(int param)
{
    return param + 137;
}
```

And then writing

```
transform(container.begin(), container.end(), container.begin(), Add137);
```

While this code works correctly, this approach is not particularly robust. What if later on we needed to increment all elements in a container by 42? Or perhaps by an arbitrary value? Ultimately, we might want to consider replacing `Add137` by a function like this one:

```
int AddTwoInts(int one, int two)
{
    return one + two;
}
```

And then using binders to lock the second parameter in place. For example, here's code that's equivalent to what we've written above:

```
transform(container.begin(), container.end(), container.begin(),
    bind2nd(ptr_fun(AddTwoInts), 137));
```

At this point, our code is completely correct, but it can get a bit annoying to have to write a function `AddTwoInts` that simply adds two integers in every program we write that needs it. Moreover, if you were to then try to increment all `double`s in a container by 1.37, you'd have to write another function `AddTwoDoubles` to avoid problems from typecasts and truncations. Fortunately, to save you the hassle of writing this code by yourself every time you need it, the STL functional library provides a large number of template adaptable function classes that simply apply the basic C++ operators to two values. These functors have names like `plus` and `divides` and accept a template type indicating what types of elements you'll be using the functor class in conjunction with. For example, in the above code, we can use the adaptable function class `plus<int>` instead of our `AddTwoInts` function, resulting in code that looks like this:

```
transform(container.begin(), container.end(), container.begin(),
    bind2nd(plus<int>(), 137));
```

Note that we need to write `plus<int>()` instead of simply `plus<int>`, since we're using the temporary object syntax to create a temporary `plus<int>` object. Forgetting the parentheses can cause a major compiler error headache that can take a while to track down.

For reference, here's a list of the common "operator functions" exported by `<functional>`:

`plus`	`multiplies`	`divides`	`minus`	`modulus`	`equal_to`
`less`	`less_equal`	`greater`	`greater_equal`	`not_equal_to`	

N-ary Functions

An *N-ary* function is simply a function that takes N parameters. For example, the library function `toupper` is a 1-ary (unary) function, while the `string`'s `insert` function has several overloads that include 2-ary (binary), 3-ary (ternary), and 4-ary functions. The adjective describing the number of parameters a function has is *arity*, as in "This function has arity 4."

The STL functional libraries only provide support for adaptable unary and binary functions, but commonly you'll encounter situations where you will need to bind and negate functions with more than two parameters. In these cases, one of your only options is to construct functor classes that accept the extra parameters in their constructors, as we covered last week.

More to Explore

Here are some topics you might want to read into if functional programming interests you. Many of these arise in professional code, so you may want to read into them a bit.

1. **`mem_fun` and `mem_fun_ref`**: If you have a container of classes, you might encounter situations where you'd like to invoke a member function of each class. For example, you might want to use `transform` to convert a `vector<string>` into a `vector<int>` of the lengths of those strings. For these scenarios, you can use the `mem_fun_ref` and `mem_fun` functions, which convert member functions into adaptable functions. `mem_fun` and `mem_fun_ref` have several idiosyncrasies, so be sure to consult a reference for more information.
2. **The Boost Libraries**: The Boost libraries have amazingly powerful support for functional programming, including rudimentary lambda expressions and support for high arity functions. If you enjoy functional programming, make sure to read into the Boost lambda libraries.
3. **Other Programming Languages**: Although this is a C++ class, if you're interested in functional programming, you might want to consider learning other programming languages like LISP, Scheme, ML, or Haskell. Functional programming is an entirely different way of thinking about programming, and if you're interested you should definitely consider expanding your horizons with other languages.

Practice Problems

Each of the problems below can be solved remarkably concisely using functional programming (in fact, the first four have one-line solutions). Try to use the `<functional>` header as much as possible in these exercises to see exactly how tight you can make your code.

1. The reciprocal of a number ***x*** is the value 1 / ***x***. Write a function `ComputeReciprocals` that accepts a `vector<double>` and returns a `vector<double>` containing the reciprocals of all of the input numbers. Do not write any functions – use the `divides` functor and a binder.
2. Write a function `ClearAllStrings` that accepts a `vector<char *>` of C strings and sets each string to be the empty string.
3. The ROT128 cipher is a weak encryption cipher that works by adding 128 to the value of each character in a string to produce a garbled string. Since `char` can only hold 255 different values, two successive applications of ROT128 will produce the original string. Write a function `ApplyROT128` that accepts a `string` and returns the `string`'s ROT128 cipher equivalent.
4. Write a template function `CapAtValue` that accepts a `vector<ElemType>` by reference and an `ElemType` by reference-to-`const` and replaces all elements in the `vector` that compare greater than the parameter with a copy of the parameter. *(Hint: use the `replace_if` algorithm)*
5. The *standard deviation* of a set of data is defined as

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}.$$

Where **N** is the number of elements in the data set, x_i represents the ith number in the data set, and $\overline{x}$ is the average of the elements in the data set. Write a function `StandardDeviation` that accepts a `vector<double>` and returns the standard deviation of the elements in the `vector`.