C++0x

Introduction

C++ is a constantly-evolving language, and over the past few years the C++ language standards body has been developing the next revision of C++, nicknamed "C++0x." C++0x is a major upgrade to the C++ programming language and as we wrap up our one-quarter tour of C++, I thought it appropriate to conclude by exploring what C++0x will have in store. This handout covers some of the more impressive features of C++0x and what to expect in the future.

Be aware that C++0x has not yet been finalized, and the material in this handout may not match the final C++0x specification. However, it should be a great launching point so that you know where to look to learn more about the next release of C++.

Automatic Type Inference

Consider the following piece of code:

```
void DoSomething(const multimap<string, vector<int> > &myMap)
{
    const pair<multimap<string, vector<int> >::const_iterator,
               multimap<string, vector<int> >::const_iterator> eq =
        myMap.equal_range("String!");
    for(multimap<string, vector<int> >::const_iterator itr = eq.first;
        itr != eq.second; ++itr)
        cout << itr->size() << endl;
}
```

This above code takes in a `multimap` mapping from `string`s to `vector<int>`s and prints out the length of all `vector`s in the `multimap` whose key is "String!" While the code is perfectly legal C++, it is extremely difficult to follow because more than half of the code is spent listing the types of two variables, `eq` and `itr`. If you'll notice, these variables can only take on one type – the type of the expression used to initialize them. Since the compiler knows all of the types of the other variables in this code snippet, couldn't we just ask the compiler to give `eq` and `itr` the right types? Fortunately, in C++0x, the answer is yes thanks to a new language feature called *type inference*. Using type inference, we can rewrite the above function in about half as much space:

```
void DoSomething(const multimap<string, vector<int>>& myMap)
{
    const auto eq = myMap.equal_range("String!");
    for(auto itr = eq.first; itr != eq.second; ++itr)
        cout << itr->size() << endl;
}
```

Notice that we've replaced all of the bulky types in this expression with the keyword `auto`, which tells the C++0x compiler that it should infer the proper type for a variable. The standard iterator loop is now considerably easier to write, since we can replace the clunky

`multimap<string, vector<int>>::const_iterator` with the much simpler `auto`. Similarly, the hideous return type associated with `equal_range` is entirely absent.

Because `auto` must be able to infer the type of a variable from the expression that initializes it, you can only use `auto` when there is a clear type to assign to a variable. For example, the following is illegal:

```
auto x;
```

Since `x` could theoretically be of any type.

`auto` is also useful because it allows complex libraries to hide information details behind the scenes. For example, recall that the `ptr_fun` function from the STL `<functional>` library takes as a parameter a function pointer and returns an adaptable version of that function. However, we never discussed what the return type of this function is; only that it returns an adaptable function. It turns out that `ptr_fun` returns an object of type `pointer_to_unary_function<Arg, Ret>` or `pointer_to_binary_function<Arg1, Arg2, Ret>`, depending on whether the parameter is a unary or binary function. This means that if you want to use `ptr_fun` to create an adaptable function and want to store the result for later use, using current of C++ you'd have to write something to the effect of

```
pointer_to_unary_function<int, bool> ouchies = ptr_fun(SomeFunction);
```

This is terribly hard to read but more importantly breaks the wall of abstraction of `ptr_fun`. The entire purpose of `ptr_fun` is to hide the transformation from function to functor, and as soon as you are required to know the return type of `ptr_fun` the benefits of the automatic wrapping facilities vanish. Fortunately, `auto` can help maintain the abstraction, since we can rewrite the above as

```
auto awwHowNice = ptr_fun(SomeFunction);
```

C++0x will provide a companion operator to `auto` called `decltype` that returns the type of a given expression. For example, `decltype(1 + 2)` will evaluate to `int`, while `decltype(new char)` will be `char *`. `decltype` does not evaluate its argument – it simply returns the compile-time type – and thus incurs no cost at runtime.

One potential use of `decltype` arises when writing template functions. For example, suppose that we want to write a template function as follows:

```
template <typename T> /* some type */ MyFunction(const T& val)
{
    return val.doSomething();
}
```

This function accepts a `T` as a template argument, invokes that object's `doSomething` member function, then returns its value (note that if the type `T` doesn't have a member function `doSomething`, this results in a compile-time error). What should we use as the return type of this function? We can't tell by simply looking at the type `T`, since the `doSomething` member function could theoretically return any type. However, by using `decltype` and a new function declaration syntax, we can rewrite this as

```
template <typename T>
auto MyFunction(const T& val) -> decltype(val.doSomething())
{
    return val.doSomething();
}
```

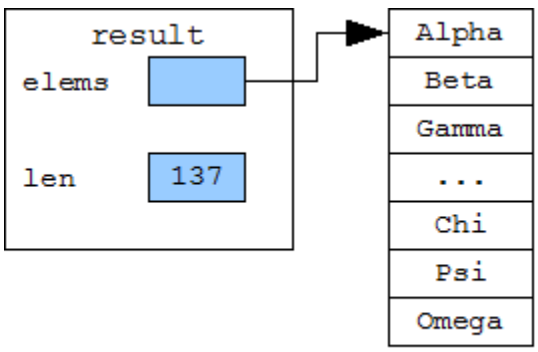
Notice that we defined the function's return type as `auto`, and then after the parameter list said that the return type is `decltype(val.doSomething())`. This new syntax for function declarations is purely optional, but will make complicated function prototypes easier to read.

Move Semantics

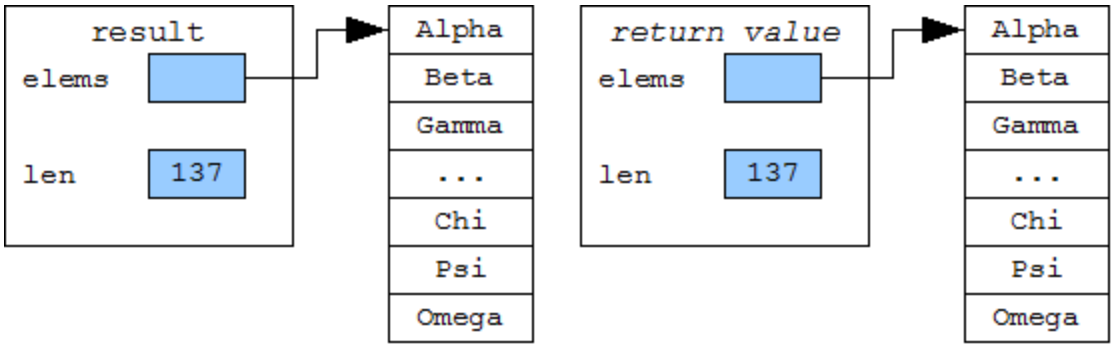
If you'll recall from our discussion of copy constructors and assignment operators, when returning a value from a function, C++ initializes the return value by invoking the class's copy constructor. While this method guarantees that the returned value is always valid, it can be grossly inefficient. For example, consider the following code:

```
vector<string> LoadAllWords(const string& filename)
{
    ifstream input(filename.c_str());
    if(!input.is_open()) throw runtime_error("File not found!");

    /* Use the vector's insert function, plus some istream_iterators, to
     * load the contents of the file.
    */
    vector<string> result;
    result.insert(result.begin(), istream_iterator<string>(input),
                  istream_iterator<string>());

    return result;
}
```

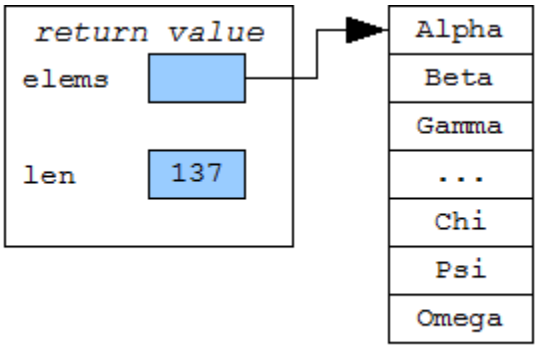
Here, we open the file specified by `filename`, then use a pair of `istream_iterator`s to load the contents of the file into the `vector`. At the end of this function, before the `return result` statement executes, the memory associated with the `result` vector looks something like this (assuming a `vector` is implemented as a pointer to a raw C++ array):



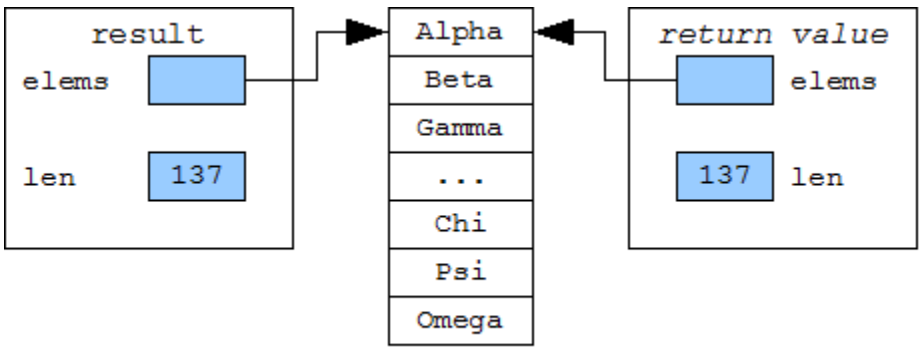
Now, the statement `return result` executes and C++ initializes the return value by invoking the `vector` copy constructor. After the copy the program's memory looks like this:



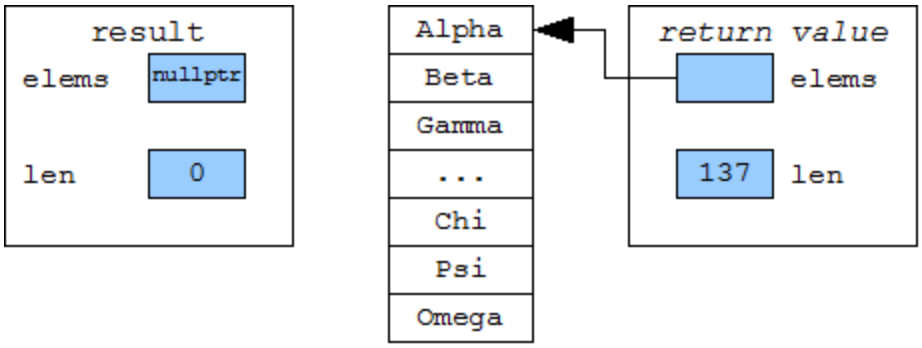
After the return value is initialized, `result` will go out of scope and its destructor will clean up its memory. Memory now looks like this:



Here, we made a full deep copy of the contents of the returned object, then deallocated all of the original memory. This is inefficient, since we needlessly copied a long list of strings. There is a much better way to return the `vector` from the function. Instead of initializing the return value by making a deep copy, instead we'll make it a shallow copy of `vector` we're returning. The in-memory representations of these two `vector`s thus look like this:



Although the two `vector`s share the same memory, the returned `vector` has the same contents as the source `vector` and is in fact indistinguishable from the original. If we then modify the original `vector` by detaching its pointer from the array and having it point to `NULL` (or, since this is C++0x, the special value `nullptr`), then we end up with a picture like this:



Now, `result` is an empty `vector` whose destructor will not clean up any memory, and the calling function will end up with a `vector` whose contents are exactly those returned by the function. We've successfully returned the value from the function, but avoided the expensive copy. In our case, if we have a `vector` of *n* `string`s of length at most *m*, then the algorithm for copying the `vector` will take O(*mn*). The algorithm for simply transferring the pointer from the source `vector` to the destination, on the other hand, is O(1) for the pointer manipulations.

The difference between the current method of returning a value and this improved version of returning a value is the difference between copy semantics and move semantics. An object has *copy semantics* if it can be duplicated in another location. An object has *move semantics* (a feature introduced in C++0x) if it can be moved from one variable into another, destructively modifying the original. The key difference between the two is the number of copies at any point. Copying an object duplicates its data, while moving an object transfers the contents from one object to another without making a copy.

To support move semantics, C++0x introduces a new variable type called an *rvalue reference* whose syntax is `Type &&`. For example, an rvalue reference to a `vector<int>` would be a `vector<int> &&`. Informally, you can view an rvalue reference as a reference to a temporary object, especially one whose contents are to be moved from one location to another.

Let's return to the above example with returning a `vector` from a function. In the current version of C++, we'd define a copy constructor and assignment operator for `vector` to allow us to return `vector`s from functions and to pass `vector`s as parameters. In C++0x, we can optionally define another special function, called a *move constructor*, that initializes a new `vector` by moving data out of one `vector` into another. In the above example, we might define a move constructor for the `vector` as follows:

```
/* Move constructor takes a vector&& as a parameter, since we want to move
 * data from the parameter into this vector.
 */
template<typename T> vector<T>::vector(vector&& other)
{
    /* We point to the same array as the other vector and have the same
     * length.
     */
    elems = other.elems;
    len = other.len;

    /* Destructively modify the source vector to stop sharing the array. */
    other.elems = nullptr;
    other.len = 0;
}
```

Now, if we return a `vector` from a function, the new `vector` will be initialized using the move constructor rather than the regular copy constructor.

We can similarly define a *move assignment operator* (as opposed to the traditional *copy* assignment operator), as shown here:

```
template<typename T> vector<T>& vector<T>::operator= (vector&& other)
{
    if(this != &other)
    {
        delete [] elems;

        /* We point to the same array as the other vector and have the same
         * length.
         */
        elems = other.elems;
        len = other.len;

        /* Modify the source vector to stop sharing the array. */
        other.elems = nullptr;
        other.len = 0;
    }
    return *this;
}
```

The similarity between a copy constructor and copy assignment operator is also noticeable here in the move constructor and move assignment operator. In fact, we can rewrite the pair using helper functions `clear` and `moveOther`:

```
template <typename T> void vector<T>::moveOther(vector&& other)
{
    /* We point to the same array as the other vector and have the same
     * length.
     */
    elems = other.elems;
    len = other.len;

    /* Modify the source vector to stop sharing the array. */
    other.elems = nullptr;
    other.len = 0;
}

template <typename T> void vector<T>::clear()
{
    delete [] elems;
    len = 0;
}

template<typename T> vector<T>::vector(vector&& other)
{
    moveOther(move(other)); // See later section for move
}
```

```
template<typename T> vector<T>& vector<T>::operator =(vector&& other)
{
    if(this != &other)
    {
        clear();
        moveOther(move(other));
    }
    return *this;
}
```

Move semantics are also quite useful in situations other than returning objects from functions. For example, suppose that we want to insert an element into an array, shuffling all of the other values down one spot to make room for the new value. Using current C++, the code for this operation is as follows:

```
template <typename T>
void InsertIntoArray(T* elems, int size, int position, const T& toAdd)
{
    for(int i = size; i > position; ++i)
        elems[i] = elems[i - 1]; // Shuffle elements down.
    elems[i] = toAdd;
}
```

There is nothing wrong *per se* with this code as it's written, but if you'll notice we're using copy semantics to shuffle the elements down when move semantics is more appropriate. After all, we don't want to *copy* the elements into the spot one element down; we want to *move* them.

In C++0x, we can use an object's move semantics (if any) by using the special helper function `move`, exported by `<utility>`, which simply returns an rvalue reference to an object. Now, if we write

```
a = move(b);
```

If `a` has support for move semantics, this will move the contents of `b` into `a`. If `a` does *not* have support for move semantics, however, C++ will simply fall back to the default object copy behavior using the assignment operator. In other words, supporting move operations is purely optional and a class can still use the old fashioned copy constructor and assignment operator pair for all of its copying needs.

Here's the rewritten version of `InsertIntoArray`, this time using move semantics:

```
template <typename T>
void InsertIntoArray(T* elems, int size, int position, const T& toAdd)
{
    for(int i = size; i > position; ++i)
        elems[i] = move(elems[i - 1]); // Move elements down.
    elems[i] = toAdd;
}
```

Curiously, we can potentially take this one step further by moving the new element into the array rather than copying it. We thus provide a similar function, which we'll call `EmplaceIntoArray`, which *emplaces* (moves) the parameter into the specified position:

```
template <typename T>
void EmplaceIntoArray(T* elems, int size, int position, T&& toAdd)
{
    for(int i = size; i > position; ++i)
        elems[i] = move(elems[i - 1]); // Move elements down.

    /* Note that even though toAdd is an rvalue reference, we still must
     * explicitly move it in.  This prevents us from accidentally using
     * move semantics in a few edge cases.
     */
    elems[i] = move(toAdd);
}
```

Move semantics and copy semantics are independent and in C++0x it will be possible to construct objects that can be moved but not copied or vice-versa. Initially this might seem strange, but there are several cases where this is exactly the behavior we want. For example, it is illegal to copy an `ofstream` because the behavior associated with the copy is undefined – should we duplicate the file? If so, where? Or should we just share the file? However, it is perfectly legitimate to *move* an `ofstream` from one variable to another, since at any instant only one `ofstream` variable will actually hold a reference to the file stream. Thus functions like this one:

```
ofstream GetTemporaryOutputFile()
{
    /* Use the tmpnam() function from <cstdio> to get the name of a
     * temporary file.  Consult a reference for more detail.
     */
    char tmpnamBuffer[L_tmpnam];
    ofstream result(tmpnam(tmpnamBuffer));
    return result; // Uses move constructor, not copy constructor!
}
```

Will be perfectly legal in C++0x because of move constructors, though the same code will not compile in current C++ because `ofstream` has no copy constructor.

Move semantics and rvalue references may seem confusing at first, but promise to be a powerful and welcome addition to the C++ family.

Lambda Expressions

In our first discussion of functional programming with the STL, we discussed how to use `count_if` to count the number of integers in a file that were less than some constant determined at runtime. Several weeks ago, we settled on two different ways of solving the problem – the first using a functor, and the second using the STL `<functional>` library. Both ways are reprinted below:

```
/* Using a functor */
struct ShorterThan
{
    explicit ShorterThan(int maxLength) : length(maxLength) {}
    bool operator() (const string &str) const
    {
        return str.length() < length;
    }
private:
    int length;
};
const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), ShorterThan(myValue));

/* Using the <functional> libraries. */
const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), bind2nd(less<int>(), myValue));
```

Both of these approaches have their weaknesses. The functor-based approach has a huge amount of boilerplate code, while the the `<functional>` approach is riddled with cryptic library functions. What we'd prefer instead is the ability to write code to this effect:

```
const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), the int is less than myValue);
```

Using a new C++0x language feature known as *lambda expressions* (a term those of you familiar with languages like Scheme, ML, or Haskell might recognize), we can write code that very closely mirrors this structure. One possibility looks like this:

```
const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), [myValue](int x) { x < myValue });
```

This rather strange construct in the final line of code is a lambda expression, an unnamed ("anonymous") function that exists only as a parameter to `count_if`. In this example, we pass as the final parameter to `count_if` a temporary function that accepts a single integer parameter and returns a `bool` indicating whether or not it is less than `myValue`. The bracket syntax `[myValue]` before the parameter declaration `(int x)` indicates to C++ that it should copy the value of `myValue` into a temporary location so that the lambda expression can access it.

Behind the scenes, C++ converts lambda expressions such as the one above into uniquely-named functors, so the above code is effectively identical to the functor-based approach outlined above.

For those of you with experience in a functional programming language, the example outlined above should strike you as an extraordinarily powerful addition to the C++ programming language. Lambda expressions greatly simplify many tasks and represent an entirely different way of thinking about programming. It will be interesting to see how rapidly lambda expressions are adopted in professional code.

Concepts

One of the most sweeping and powerful features introduced in C++0x are a pair of language constructs called *concepts* and *concept maps*. To understand the motivation underlying concepts, consider the following implementation of the `copy` algorithm:

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator start, InputIterator stop,
                    OutputIterator out)
{
    for(; start != stop; ++start; ++out)
        *out = *start;
    return out;
}
```

Here, we simply walk the `start` iterator forward until it hits the `stop` iterator, storing in the element referenced by `out` the value currently being iterated over.

Notice that this function is templatized over two types named `InputIterator` and `OutputIterator`. As we mentioned in our discussion of functors, these names have absolutely no bearing about what types *can* be passed as parameters and serve solely as indicators about what sorts of objects *should* be passed as parameters. Provided that the programmer passes valid input and output iterators to `copy`, the function will compile and work correctly. If we try passing in nonsense values to `copy`, C++ will report these errors as errors in the body of the template, rather than as bad parameters to the function. For example, try compiling this code:

```
copy(0, 0, 137);
```

Microsoft Visual Studio 2003 reports several error messages about "illegal indirection" (dereferencing something that isn't a pointer) rather than a more informative error message informing me that I shouldn't be passing in integer parameters to `copy`. What if there was some way to tell C++ that a certain template function can only accept a type as a parameter if it meets certain criteria? This currently cannot be accomplished in C++, but using C++0x concepts, we can define exactly what behavior we want to have available in a template type.

In C++0x, a *concept* is a specification that defines what functionality must be available in a given type in order to pass it as a parameter to a templatized class or function. For example, here's a concept that specifies that a type must have a function called `doSomething` that takes no arguments:

```
auto concept MustDoSomething<typename T>
{
    void T::doSomething() const;
};
```

Here, the declaration `void T::doSomething() const` means that if a type *conforms* to the `MustDoSomething` concept, it must support a `doSomething` member function that accepts nothing and is `const`. The declaration `auto concept MustDoSomething<typename T>` defines a new concept called `MustDoSomething` that's parametrized over some type `T`, and the `auto` keyword indicates that any type that supports `doSomething` can automatically be treated as a `MustDoSomething`.

Now, let's suppose that we want to write a template function that accepts a parameter only if it exports a `doSomething` function. Using current C++ templates, we'd write it as follows:

```
template<typename T> void HaveSomethingDoSomething(const T& elem)
{
    elem.doSomething();
}
```

Using C++0x concepts, we can rewrite the above as

```
template<MustDoSomething T> void HaveSomethingDoSomething(const T& elem)
{
   elem.doSomething();
}
```

Note that instead of specifying that `T` is a `typename`, we've explicitly indicated that it's a `MustDoSomething`. This has several advantages. First, if we try passing a parameter to `HaveSomethingDoSomething` that doesn't conform to the `MustDoSomething` concept, the compiler can report that the error is in the template argument, rather than in the body of the template. Second, by explicitly indicating that the parameter must conform to `MustDoSomething`, programmers not familiar with the template function can get a better idea of what that function expects of its parameters, increasing readability. Finally, and perhaps most importantly, because concepts give explicit restrictions on the interface of a template argument, the C++ compiler will be able to rigorously type-check the body of a constrained template function (a template function where all of the parameters are restricted by some sort of concept) without ever having to instantiate the template. For example, consider the following two pieces of code:

```
template<typename T> void HaveSomethingDoSomethingOld(const T& elem)
{
    elem.doSomething(137); // Pass in nonexistent parameter
}

template<MustDoSomething T> void HaveSomethingDoSomethingNew(const T& elem)
{
    elem.doSomething(137); // Pass in nonexistent parameter
}
```

Suppose that we want to instantiate both of these templates using a type that conforms to the `MustDoSomething` concept. The first function, which uses only regular C++ templates, will compile without warnings or errors unless we actually call the template function in our code. On the other hand, because `HaveSomethingDoSomethingNew` has its parameter constrained by the `MustDoSomething` concept, C++0x can analyze the body of the function and detect that misuse of the `elem` parameter. This will greatly simplify library implementation.

In the above example with `MustDoSomething`, we specified that the concept was an `auto` concept, meaning that any class that has a `doSomething` method with the proper signature can be used whenever a `MustDoSomething` is expected. However, there is a second type of concept, known as a “plain concept,” that lacks the `auto` keyword. For example, consider the following concept:

```
concept Range<typename ContainerType>
{
    typename iterator; // There must be an iterator type for the range
    iterator begin(ContainerType); // Given a container, return start iterator
    iterator end(ContainerType); // Given a container, return stop iterator
};
```

Note that this concept is not marked `auto`, which means that given a type that has a nested `iterator` type along with a `begin` and `end` function, we still cannot use that type wherever `Range` is expected. In order to make a type conform to the `Range` concept, we'll use a *concept map*, which defines how a given type conforms to a plain concept. For example, suppose that we want to be able to use a `vector` whenever a `Range` is expected. Then we can write the following:

```
/* For any type, a vector of that type can be converted into a Range. */
template <typename ElemType> concept_map Range<vector<ElemType>>
{
    /* The iterators are vector iterators. */
    typedef vector<ElemType>::iterator iterator;

    /* begin and end just call begin and end. */
    iterator begin(const vector<ElemType>& vec)
    {
        return vec.begin();
    }
    iterator end(const vector<ElemType>& vec)
    {
        return vec.end();
    }
}
```

There are several points to note here. First, the entire `concept_map` itself is a template, since `vector` is a template and we'd like any type of `vector` to be used whenever `Range` is defined. Next, notice that for each trait specified in the concept `Range`, we have a corresponding definition inside this `concept_map`. For example, in the `concept` we specified that there must be some `typename iterator`, and in the `concept_map` for `vector<ElemType>` we've defined this type `iterator` by `typedef`ing the `vector`'s iterators as `iterator`. Also, note that inside this `concept_map`, we've defined `begin` and `end` as functions that accept a `vector` and return its `begin` and `end` values. These functions are defined inside the `concept_map`, so if we pass a `vector` to a template function expecting a `Range`, if the template function calls `begin` or `end`, it will call these functions specified here. To see how this all works together, consider the following code:

```
template <Range R> void PrintRange(const R& range)
{
    for(auto itr = Range<R>::begin(range); itr != Range<R>::end(range); ++itr)
        cout << *itr << endl;
}
```

Here, we've defined a `PrintRange` function that accepts an object that conforms to the `Range` concept. Note that the parameter itself is *not* the range – the fact that the parameter conforms to `Range` simply indicates that the type of the parameter permits us to *extract* a range. Next, in the body of the function, we use a for loop to traverse the elements. As our start point, we use `Range<R>::begin(range)`. The syntax `Range<R>` instructs C++0x to look up the `concept_map` associated with the type `R`, then to call its `begin` function. We continue looping until we hit `Range<R>::end(range)` and continue stepping

forward by writing `++itr`. The beauty of this whole system is that we don't care how this range is defined – it could be from a container, a `pair` of iterators, or even a `pair<int, int>`s with a `concept_map` that maps the pair into all of the integer values between the `first` and `second` fields.

Interestingly, the `Range` concept illustrated above is very similar to a `std::Range` concept introduced in the new standard. `std::Range` is a concept similar to the one above that enables the programmer to write a special type of for loop over the elements of the range. The syntax is

```
for(element : container) { /* ... */ }
```

For example, in C++0x we can write

```
for(const auto &elem: myVector)
    cout << elem << endl;
```

Library Extensions

In addition to all of the language extensions mentioned in the above sections, C++0x will provide a new set of libraries that should make certain common tasks much easier to perform:

- **Enhanced Smart Pointers**. C++0x will support a wide variety of smart pointers, such as the reference-counted `shared_ptr` and the aforementioned `unique_ptr`.
- **New STL Containers**. The current STL associative containers (`map`, `set`, and the multicontainers) are layered on top of balanced binary trees, which means that traversing the `map` and `set` always produce elements in sorted order. However, the sorted nature of these containers means that insertion, lookup, and deletion are all O(lg *n*), where *n* is the size of the container. In C++0x, the STL will be enhanced with `unordered_map`, `unordered_set`, and multicontainer equivalents thereof. These containers are layered on top of hash tables, which have O(1) lookup and are extremely useful when ordering is not important.
- **Multithreading Support**. Virtually all major C++ programs these days contain some amount of multithreading and concurrency, but the C++ language itself provides no support for concurrent programming. The next incarnation of C++ will support a threading library, along with atomic operations, locks, and all of the bells and whistles needed to write robust multithreaded code.
- **Regular Expressions**. The combination of C++ `string`s and the STL algorithms encompasses a good deal of string processing functionality but falls short of the features provided by other languages like Java, Python, and (especially) Perl. C++0x will augment the strings library with full support for regular expressions, which should make string processing and compiler-authoring considerably easier in C++.
- **Random Number Generation**. C++'s only random number generator is `rand`, which has extremely low randomness (on some implementations numbers toggle between even and odd) and is not particularly useful in statistics and machine learning applications. C++0x, however, will support a rich random number generator library, complete with a host of random number generators and probability distribution functors.
- **Concepts and Generic Programming**. C++0x will provide a staggeringly large number of useful concepts that can help generic programmers write extremely optimized code. Want to know if a template argument has a copy constructor? Just check if it conforms to `std::CopyConstructible`.

Other Key Language Features

In addition to the language features listed above, there are many other upgrades you might find useful. Here's a small sampling:

- **Variadic Templates**: Templates that can take an arbitrary number of arguments. Useful for writing generalized tuple types.
- **Unified Initialization Syntax**: It will be possible to initialize C++ classes by using the curly brace syntax (e.g. `vector<int> v = {1, 2, 3, 4, 5};`)
- **Delegating Constructors**: Currently, if several constructors all need to access the same code, they must call a shared member function to do the work. In C++0x, constructors can invoke each other in initializer lists.
- **Explicit Copying/Assignment Control**: It will be possible to explicitly state that a class does not have a copy constructor or assignment operator, instead of having to cram the functions into a base class or private section.
- **Better Enumerations**: Currently, `enum` can only be used to create integral constants, and those constants can be freely compared against each other. In C++0x, you will be able to specify what type to use in an enumeration, and can disallow automatic conversions to `int`.
- **Angle Brackets**: It is currently illegal to terminate a nested template by writing two close brackets consecutively, since the compiler confuses it with the stream insertion `operator >>`. This will be fixed in C++0x.
- **C99 Compatibility**: C++0x will formally introduce the `long long` type, which many current C++ compilers support, along with various preprocessor enhancements.

Final Thoughts

It's been quite a trip since we first started with the IOStream library ten weeks ago. You now know how to program with the STL, write well-behaved C++ objects, and even how to use functional programming constructs inside the otherwise imperative/object-oriented C++ language. But despite the immense amount of material we've covered this quarter, we have barely scratched the surface of C++. There are volumes of articles and books out there that cover all sorts of amazing C++ tips and tricks, and by taking the initiative and exploring what's out there you can hone your C++ skills until problem solving in C++ transforms from “how do I solve this problem?” to “which of these many options is best for solving this problem?”

C++ is an amazing language. It has some of the most expressive syntax of any modern programming language, and affords an enormous latitude in programming styles. Of course, it has its flaws, as critics are eager to point out, but despite the advent of more modern languages like Java and Python C++ still occupies a prime position in the software world.

I hope that you've enjoyed CS106L as much as I have. If you have any comments, suggestions, or criticisms, please be sure to include them in your end-of-quarter evaluation. Like the C++ language, this class is constantly evolving, and if there's anything I can do to make the class more enjoyable, be sure to let me know!

Have fun with C++, and I wish you the best of luck wherever it takes you!