Chapter 14: C++0x

C++0x feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever. If you timidly approach C++ as just a better C or as an object-oriented language, you are going to miss the point. The abstractions are simply more flexible and affordable than before. Rely on the old mantra: If you think [o]f it *as a separate idea or object, represent it directly in the program; model real-world objects, concepts, and abstractions directly in code. It's easier now: Your ideas will map to enumerations, objects, classes (e.g. control of defaults), class hierarchies (e.g. inherited constructors), templates, concepts, concept maps, axioms, aliases, exceptions, loops, threads, etc., rather than to a single "one size fits all" abstraction mechanism.*

My ideal is to use programming language facilities to help programmers think differently about system design and implementation. I think C++0x can do that – and do it not just for C++ programmers but for programmers used to a variety of modern programming languages in the general and very broad area of systems programming.

In other words, I'm still an optimist.

– Bjarne Stroustrup, inventor of C++. [Str09.3]

C++ is constantly evolving. Over the past few years the C++ 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 tour of C++, I thought it appropriate to conclude by exploring what C++0x will have in store. This chapter 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 chapter 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 vectors 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 implementation details behind-the-scenes. For example, recall that the `ptr_fun` function from the STL `<functional>` library takes as a parameter a regular C++ function and returns an adaptable version of that function. In our discussion of the library's implementation, we saw that the return type of `ptr_fun` is either `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 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 howNice = 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 yields its 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();
}
```

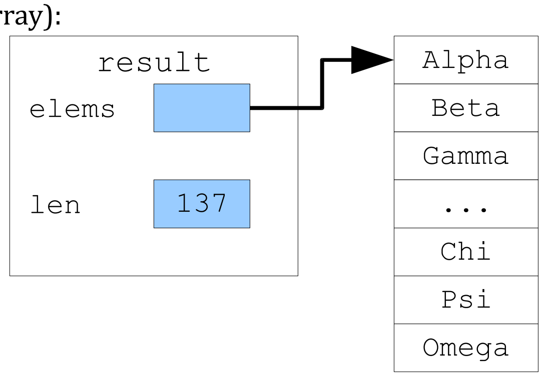
Notice that we defined the function's return type as `auto`, and then after the parameter list said that the return type is `decltype(val.doSomething())`. This new syntax for function declarations is optional, but will make complicated function prototypes easier to read.

Move Semantics

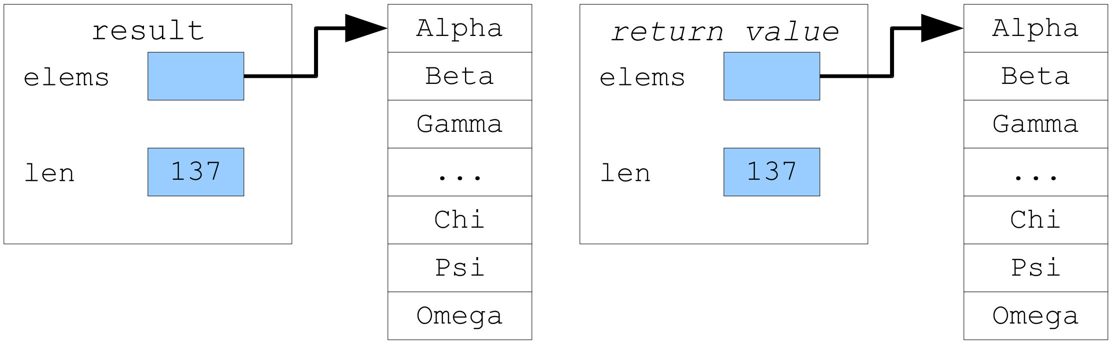
If you'll recall from our discussion of copy constructors and assignment operators, when returning a value from a function, C++ initializes the return value by invoking the class's copy constructor. While this method guarantees that the returned value is always valid, it can be grossly inefficient. For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

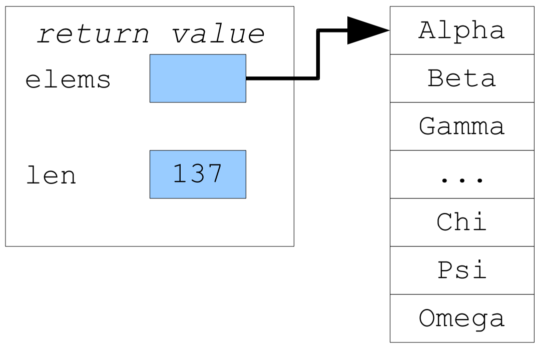
Here, we open the file specified by `filename`, then use a pair of `istream_iterator`s to load the contents of the file into the `vector`. At the end of this function, before the `return result` statement executes, the memory associated with the `result vector` looks something like this (assuming a `vector`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



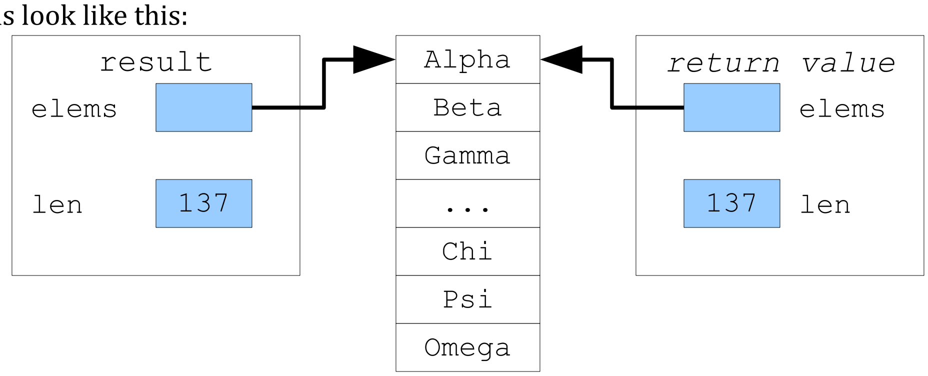
Now, the statement `return result` executes and C++ initializes the return value by invoking the `vector` copy constructor. After the copy the program's memory looks like this:



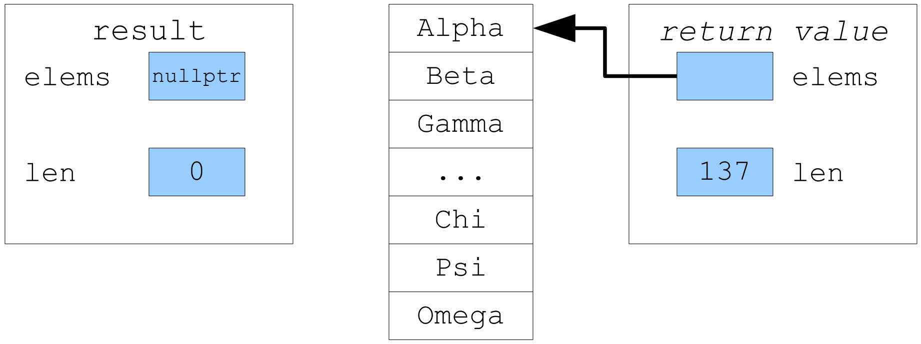
After the return value is initialized, `result` will go out of scope and its destructor will clean up its memory. Memory now looks like this:



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 vectors thus look like this:



Although the two vectors 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 other 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;

        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 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 `MoveIntoArray`, which moves the parameter into the specified position:

```
template <typename T>
void MoveIntoArray(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.

Another example of an object that has well-defined move behavior but no copy behavior is the C++ `auto_ptr` class. If you'll recall, assigning one `auto_ptr` to another destructively modifies the original `auto_ptr`. This is exactly the definition of move semantics. However, under current C++ rules, implementing `auto_ptr` is extremely difficult and leads to all sorts of unexpected side effects. Using move constructors, however, we can eliminate these problems. C++0x will introduce a replacement to `auto_ptr` called `unique_ptr` which, like `auto_ptr`, represents a smart pointer that automatically cleans up its underlying resource when it goes out of scope. Unlike `auto_ptr`, however, `unique_ptr` cannot be copied or assigned but can be moved freely. Thus code of this sort:

```
unique_ptr<int> myPtr(new int);
unique_ptr<int> other = myPtr; // Error!  Can't copy unique_ptr.
```

Will not compile. However, by explicitly indicating that the operation is a move, we can transfer the contents from one `unique_ptr` to another:

```
unique_ptr<int> myPtr(new int);
unique_ptr<int> other = move(myPtr); // Legal; myPtr is now empty
```

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

Last chapter, we considered the problem of counting the number of `string`s in a `vector` whose lengths were less than some value determined at runtime. We explored how to solve this problem using the `count_if` algorithm and a functor. Our solution was as follows:

```
class ShorterThan {
public:
    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));
```

This functor-based approach works correctly, but has a huge amount of boilerplate code that obscures the actual mechanics of the solution. 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 string is shorter 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](const string& x) { return x.length() < myValue; });
```

The 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 `string` parameter and returns a `bool` indicating whether or not its length is less than `myValue`. The bracket syntax `[myValue]` before the parameter declaration `(int x)` is called the *capture list* and indicates to C++ that the lambda expression can access the value of `myValue` in its body.

Behind the scenes, C++ converts lambda expressions such as the one above into uniquely-named functors, so the above code is 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.

Variadic Templates

In the previous chapter we implemented a class called `Function` that wrapped an arbitrary unary function. Recall that the definition of `Function` is as follows:

```
template <typename ArgType, typename ReturnType> class Function {
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (ArgType value) const;
private:
    /* ... */
};
```

What if we want to generalize `Function` to work with functions of arbitrary arity? That is, what if we want to create a class that encapsulates a binary, nullary, or ternary function? Using standard C++, we could do this by introducing new classes `BinaryFunction`, `NullaryFunction`, and `TernaryFunction` that were implemented similarly to `Function` but which accepted a different number of parameters. For example, here's one possible interface for `BinaryFunction`:

```
template <typename ArgType1, typename ArgType2, typename ReturnType>
class BinaryFunction {
public:
    /* Constructor and destructor. */
    template <typename BinaryFn> BinaryFunction(BinaryFn);
    ~BinaryFunction();

    /* Copy support. */
    BinaryFunction(const BinaryFunction& other);
    BinaryFunction& operator= (const BinaryFunction& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (ArgType1 arg2, ArgType2 arg2) const;
private:
    /* ... */
};
```

Writing different class templates for functions of each arity is troublesome. If we write `Function`-like classes for a fixed number of arities (say, functions between zero and ten arguments) and then discover that we need a wrapper for a function with more arguments, we we'll have to write that class from scratch. Moreover, the structure of each function wrapper is almost identical. Compare the `BinaryFunction` and `Function` class interfaces mentioned above. If you'll notice, the only difference between the classes is the number of template arguments and the number of arguments to `operator()`. Is there some way that we can use this commonality to implement a single class that works with functions of arbitrary arity? Using the current incarnation of C++ this is not possible, but using a C++0x feature called *variadic templates* we can do just this.

A *variadic template* is a template that can accept an arbitrary number of template arguments. These arguments are grouped together into arguments called *parameter packs* that can be expanded out to code for each argument in the pack. For example, the following class is parameterized over an arbitrary number of arguments:

```
template <typename... Args> class Tuple {
    /* ... */
};
```

The syntax `typename... Args` indicates that `Args` is a parameter pack that represents an arbitrary number of arguments. Since `Args` represents a list of arguments rather than an argument itself, it is illegal to use `Args` in an expression by itself. Instead, `Args` must be used in a *pattern expression* indicating what operation should be applied to each argument in the pack. For example, if we want to create a constructor for `Tuple` that accepts a list of arguments with one argument for each type in `Args`, we could write the following:

```
template <typename... Args> class Tuple {
public:
    Tuple(const Args& ...);
};
```

Here, the syntax `const Args& ...` is a pattern expression indicating that for each argument in `Args`, there should be a parameter to the constructor that's passed by reference-to-`const`. For example, if we created a `Tuple<int>`, the constructor would be `Tuple<int>(const int&)`, and if we create a `Tuple<int, double>`, it would be `Tuple<int, double>(const int&, const double&)`.

Let's return to the example of `Function`. Suppose that we want to convert `Function` from encoding a unary function to encoding a function of arbitrary arity. Then we could change the class interface to look like this:

```
template <typename ReturnType, typename... ArgumentTypes> class Function {
public:
    /* Constructor and destructor. */
    template <typename Callable> Function(Callable);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (ArgumentTypes... args) const;
private:
    /* ... */
};
```

`Function` is now parameterized such that the first argument is the return type and the remaining arguments are argument types. For example, a `Function<int, string>` is a function that accepts a `string` and returns an `int`, while a `Function<bool, int, int>` would be a function accepting two `int`s and returning a `bool`.

We've just seen how the interface for `Function` looks with `variadic` templates, but what about the implementation? If you'll recall, the original implementation of `Function`'s `operator()` function looked as follows:

```
template <typename ArgType, typename ReturnType>
ReturnType Function<ArgType, ReturnType>::operator()(ArgType param) const {
    return function->execute(param);
}
```

Let's begin converting this to use variadic templates. The first step is to adjust the signature of the function, as shown here:

```
template <typename RetType, typename... ArgTypes>
RetType Function<RetType, ArgTypes...>::operator()(ArgTypes... args) const {
    /* ... */
}
```

Notice that we've specified that this is a member of `Function<RetType, ArgTypes...>`.

In the unary version of `Function`, we implemented `operator()` by calling a stored `function` object's `execute` member function, passing in the parameter given to `operator()`. But how can we now call `execute` passing in an arbitrary number of parameters? The syntax for this again uses `...` to tell C++ to expand the `args` parameters to the function into an actual list of parameters. This is shown here:

```
template <typename RetType, typename... ArgTypes>
RetType Function<RetType, ArgTypes...>::operator()(ArgTypes... args) const {
    return function->execute(args...);
}
```

Just as using `...` expands out a parameter pack into its individual parameters, using `...` here expands out the variable-length argument list `args` into each of its individual parameters. This syntax might seem a bit tricky at first, but is easy to pick up with practice.

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`, etc.) 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 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++.

- **Upgraded `<functional>` library**. C++0x will expand on `<functional>` with a generic `function` type akin to the one described above, as well as a supercharged `bind` function that can bind arbitrary parameters in a function with arbitrary values.

- **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.

- **Metaprogramming Traits Classes**. C++0x will provide a large number of classes called *traits classes* that can help generic programmers write optimized code. Want to know if a template argument is an abstract class? Just check if `is_abstract<T>::type` evaluates to `true_type` or `false_type`.

Other Key Language Features

Here's a small sampling of the other upgrades you might find useful:

- **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.
- **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.

C++0x Today

Although C++0x has not yet been adopted as a standard, there are several freely-available compilers that support a subset of C++0x features. For example, `g++` versions 4.4 and up have support for much of C++0x, and Microsoft Visual Studio 2010 has a fair number of features implemented, including lambda expressions and the `auto` keyword. If you want to experience the future of C++ today, consider downloading one of these compilers.