C++0x: The future of C++

By Alex Allain

What is C++0x?

C++0x was the working name for the new standard for C++, adding many language features that I'll cover in this series on C++11. In September 2011, C++0x was officially published as the new C++11 standard, and many compilers now provide support for some of the core C++11 features.

C++11 includes a wide range of features: major new features like lambda support and "move semantics", usability improvements like type inference through the auto keyword, simplified looping over containers, and many improvements that will make templates easier to write and easier to use. This series on C++11 will cover all of these features and many more.

Should you care about C++11?

Most definitely. C++11 adds many new language features to C++. C++11 should fix many annoyances and reduce the overall verbosity of C++ as well as provide new tools, such as lambda expressions, that increase its overall expressiveness and clarity. Fetaures like move semantics improve the basic efficiency of the language, allowing you to write faster code, and the improvements to the template system make it much easier to write generic code.

The new standard library will also include many new features, including adding multithreading support directly into C++ and improved smart pointers that will simplify memory management for those who aren't already using features like boost::shared_ptr.

I've started using several new C++11 features professionally and I'm loving it. Some of the new features I'm fond of include the new meaning of the auto keyword, simplifications like better handling of right angle brackets in templates, lambda expressions and the new function declaration syntax.

How was C++11 developed?

I can't go on further about C++11 without acknowledging the hard work done by the C++ standards committee--a group of experts from academia and industry who have met many times to work through all the edge cases and design a programming language that can be implemented across multiple platforms by multiple compilers, producing efficient and reasonably maintainable code. The next standard, C++11, looks to be a fantastic addition to the flexibility and power of C++.

What is C++11 about?

Language usability

Having started to use C++11, I'd say that the most fundamental way of looking at it is that it makes C++ a much more usable language. This isn't to say that it makes it a simpler language--there are lots of new features--but it provides a lot of functionality that makes it easier to program. Let's look at one example, the auto keyword.

In C++11, if the compiler is able to determine the type of a variable from its initialization, you don't need to provide the type. For example, you can write code such as

```
int x = 3;
auto y = x;
```

and the compiler will deduce that y is an int. This, of course, isn't a shining example of where auto is really useful. Auto really comes into its own when working with templates and especially the STL. Why is that? Imagine working with an iterator:

```
map<string, string> address_book;
address_book[ "Alex" ] = "webmaster@cprogramming.com";
// add a bunch of people to address_book
```

Now you want to iterate over the elements of the address_book. To do it, you need an iterator:

```
map<string, string>::iterator itr = address_book.begin();
```

That's an awfully long type declaration for something that you already know the type of! Wouldn't it be nice to simply write:

```
auto itr = address_book.begin();
```

The code is much shorter, and frankly, I think it's more readable, not less, because the template syntax obscures everything else on that line. This is one of my favorite new features, and I find it eliminates a lot of headaches and hard-to-track-down compiler errors, and just generally saves time without losing expressiveness.

Ranged For Loops

Now, the iterator example is one where C++11 has come up with an even better way of handling this--something called a range-based for loop (which almost every language has nowadays). The idea is so elegant, an example should suffice:

```
vector<int> vec;
vec.push_back( 10 );
vec.push_back( 20 );

for (int &i : vec )
{
        cout << i;
}
```

All you need to do is give your variable and the range to iterate over (defined as something with iterators available via calls to begin and end--so all STL containers) and you're set! This is a pretty new feature, available as far as I know only in GCC 4.6.

But what if you want to iterate over a map? How do you put in the type for a value stored in a map? With a vector, you know the value is an int. With a map, it's essentially a pair, with .first and .second giving you the key and value. But with auto, we don't need to worry about getting the exact type right, you can simply do this:

```
for ( auto address_entry : address_book )
{
        cout  << address_entry.first << " < " << address_entry.second << ">" <<endl;
}
```

Which prints out as:

```
Alex <webmaster@cprogramming.com>
```

Isn't that a nice combination of new features in C++11? It feels like it was designed that way :)

Right angle brackets

And I have one more usability improvement for you--in previous versions of the C++ standard, if you wrote a template that had another template:

```
vector<vector<int> > vector_of_int_vectors;
```

You had to write a space between the two closing angle brackets. This was not only annoying, but if you did write >> without a space, you'd get obtuse and confusing compiler error messages. The reason for this behavior had to do with an obscure C++ lexer trait called the maximal munch rule. The good news is, you no longer need to worry about it! Say hello to

```
vector<vector<int>> vector_of_int_vectors;
```

True, this seems like a small thing, but it's a victory of human code writers over machine tools. Plus it's much less ugly. Compiler support for right-angle brackets is great: GCC since 4.3, MSVC since version 8 (!) and the Intel compiler since version 11.

Mulithreading

For the first time, the C++11 standard will include a memory model and corresponding libraries for multithreading, meaning that you'll be able to write standards-compliant multithreading code. The new standard will provide for all the normal threading functionality, such as threads and thread-local storage and atomic operations. It will also include an interesting set of features, futures and promises. The basic idea of futures and promises is that you can write code that says, "this object, a future, stands for a result that hasn't been computed yet" and the work to compute the value can take place in the background. When the value is needed, you ask the future for it; if the value is ready, you get it; if not, you wait.

I'll go into more depth on mulithreading in a later article in this series.

Lots of other stuff

The number of features in C++11 is incredibly exciting. You can get a taste for what's available on the C++11 page at Wikipedia, and I plan to dive into many of these features in more depth in this series, including:

- How auto, decltype, and the new function syntax work together to create better code
- Lambda functions
- Usability improvements like range-based for loops
- Performance improvements like generalized constant expressions
- Performance improvements like rvalue references and move semantics
- Type safety improvements like nullptr and strongly typed enumerations
- Language expressiveness improvements like the ability to explicitly override or make "final" virtual functions
- Improvements to object construction, like initialization lists, delegating constructors and explicit control over auto-generated functions
- Template improvements including templated typedefs, variadic templates, uniform initialization and static assertions
- The new C++ memory model and the feature it supports: multithreading
- Improvements to the standard library including regular expressions, hashtables and improved smart pointers
- What features were removed or deprecated by the standard, and why

Compiler Support for C++11

Of course, no language feature matters if it's not available to use, and the good news is that many compilers now support the new C++11 features. The Apache foundation has compiled a very useful list of C++11 language features and the compilers that support them: Compiler Support for C++11. If you're interested in GCC, this page describes the GCC 4.7 support for C++11.

Some compilers, such as GCC, do not automatically enable support for these features--for example, to enable C++11 features, you must compile with -std=c++0x. Nonetheless, they are still valuable if you're working on a project where you can control the choice of compiler and set of language features.

Keeping up with the news

To the the latest updates on C++11, you can subscribe to our RSS feed, or join me over on Facebook or twitter.

Next: How auto, decltype, and the new function syntax work together to create better code Learn about some of the new type inference features of C++11

Improved Type Inference in C++11: auto, decltype, and the new function declaration syntax

By Alex Allain

Note: C++11 is the now offical name for the next version of the C++ standard, previously known as C++0x.

C++11 introduces several new handy-dandy type inference capabilities that mean you can spend less time having to write out things the compiler already knows. There are, of course, times when you need to help out the compiler or your fellow programmers. But with C++11, you can spend less time on the mundane stuff and focus on the logic.

Let's start by looking at the most immediately obvious new benefit, the auto keyword.

The joy of auto

A quick refresher in case you didn't read about auto in the first article on C++0x. In C++11, if the compiler can infer the type of a variable at the point of declaration, instead of putting in the variable type, you can just write auto:

```
int x = 4;
```

can now be replaced with

```
auto x = 4;
```

This, of course, is not the intended use of auto at all! It really shines when working with templates and iterators:

```
vector<int> vec;
auto itr = vec.iterator(); // instead of vector<int>::iterator itr
```

There are other times where auto comes in handy, too. For example, let's say that you had some code of the form:

```
template <typename BuiltType, typename Builder>
void
makeAndProcessObject (const Builder& builder)
{
    BuiltType val = builder.makeObject();
    // do stuff with val
}
```

In this code, you can see that there are two necessary template parameters--one for the type of the "builder" object, and a second for the type of the object being built. Even worse, the type of the built object cannot be deduced by the template parameter. Every call must look like this:

```
MyObjBuilder builder;
makeAndProcessObject<MyObj>( builder );
```

But auto immediately reduces this ugliness to nothing because you no longer need to be able to write the specific type at the point where you build the object. You can let C++ do it for you:

```
template <typename Builder>
```

```
makeAndProcessObject (const Builder& builder)
{
    auto val = builder.makeObject();
    // do stuff with val
}
```

Now you only need a single template parameter, and that parameter is easily inferred when calling the function:

```
MyObjBuilder builder;
makeAndProcessObject( builder );
```

This is way better for the caller, and the template code loses nothing in readability--if anything, it's easier to understand!

The joy of decltype and the new return value syntax

Now you might be saying here--okay, that's great, but what if I wanted to *return* the value that this builder object created? I still need to provide the template argument becuase I need to provide the return type. Well, it turns out that the standards committee is full of smart people, and they have an extremely nice solution to this problem. It comes in two parts: decltype and the new return value syntax.

New Return Value Syntax

Let's start off with the new, optional, return value syntax, since it manages to find yet another use for auto. In all prior versions of C and C++, the return value of a function absolutely had to go before the function:

```
int multiply (int x, int y);
```

In C++11, you can now put the return value at the end of the function declaration, substituting auto for the name of the return type, if you want to:

```
auto multiply (int x, int y) -> int;
```

So would you want to do this? Let's look at a simple example where it helps us: a class with an enum declared inside it:

```
class Person
{
public:
    enum PersonType { ADULT, CHILD, SENIOR };
    void setPersonType (PersonType person_type);
    PersonType getPersonType ();
private:
    PersonType _person_type;
};
```

Here we have a simple class, Person, that has a type: whether the person is an adult, a child, or a senior citizen. Not much special about it, but what happens when you go to define the methods?

The first method, the setter, is trivial to declare, you can use the enum type PersonType without any trouble:

```
void Person::setPersonType (PersonType person_type)
{
    _person_type = person_type;
}
```

On the other hand, the second method is a bit of a mess. The nice clean looking code won't compile:

```
// the compiler doesn't know what PersonType is because PersonType is being used outside of the
// Person class
PersonType Person::getPersonType ()
{
    return _person_type;
}
```

You have to write:

```
Person::PersonType Person::getPersonType ()
{
    return _person_type;
}
```

to make the return value work correctly. This isn't that big of a deal, but it's pretty easy to do by mistake, and it can get much messier when templates are involved.

This is where the new return value syntax comes in. Because the return value goes at the end of the function, instead of before it, you don't need to add the class scope. By the point the compiler reaches the return value, it already knows the function is part of the Person class, so it knows what PersonType is.

```
auto Person::getPersonType () -> PersonType
{
    return _person_type;
}
```

Okay, while that's very nice, does it really help us out? We can't use this new syntax to solve the problem we had before, can we? Well, not yet. Let's add one more concept: decltype

decltype

Decltype is auto's not-evil twin. Auto lets you declare a variable with a particular type; decltype lets you extract the type from a variable (or any other expression). What do I mean?

```
int x = 3;
decltype(x) y = x; // same thing as auto y = x;
```

You can use decltype for pretty much any expression, including to express the type for a return value from a method. Hmm, that sounds like a familiar problem doesn't it? What if we could write:

```
decltype( builder.makeObject() )
```

This would give us the type that is returned from makeObject, allowing us to specify the return value from makeAndProcessObject. We can combine this with the new return value syntax to produce this method:

```
template <typename Builder>
auto
makeAndProcessObject (const Builder& builder) -> decltype( builder.makeObject() )
{
    auto val = builder.makeObject();
    // do stuff with val
    return val;
}
```

This only works with the new return value syntax because under the old syntax, we couldn't refer to the function argument, builder, at the point where we declare the return type. With the new syntax, all of the arguments to a function are fair game!

Auto, References, Pointers and Const

Auto, References, Pointers and Const

One question that is sure to come up is how auto handles references:

```
int& foo();

auto bar = foo(); // int& or int?
```

The short answer is in C++11, auto defaults to being by-value for references, so in the above code bar is an int. However, you can add the & as a modifier to force it to be a reference:

```
int& foo();

auto bar = foo(); // int
auto& baz = foo(); // int&
```

On the other hand, if you have a pointer auto will automatically pick up pointerness:

```
int* foo();

auto p_bar = foo(); // int*
```

But you can also (thankfully) be explicit about it, and indicate that the variable is a pointer:

```
int* foo();
auto *p_baz = foo(); // int*
```

You can similarly tack const onto auto if you need it, when dealing with references:

```
int& foo();

const auto& baz = foo(); // const int&
```

Or with pointers:

```
int* foo();
const int* const_foo();
const auto* p_bar = foo(); // const int*
auto p_bar = const_foo(); // const int*
```

Overall, it feels quite natural and normal, and it follows the normal type inference rules of templates in C++.

So, do compilers support this stuff?

As of this writing, GCC 4.4 and MSVC 10 both support everything I've talked about this article, and I'm used most of these techniques in production code; these aren't theoretical benefits, they're real. So if you're compiling your code with -std=c++0x in GCC or using MSVC 10, you can start using these techniques today. If you're using another compiler, check out this page for details of C++11 compiler support. Since the standard has been ratified, and should be published within weeks, now's the time to start.

Next: Lambda Functions in C++11 - the definitive guide One of the most exciting features of C++11 is ability to create lambda functions, learn what they are and how to use them
Previous: What is C++11 Get introduced to what C++11 means, and why you should be excited about it

Lambda Functions in C++11 - the Definitive Guide

By Alex Allain

λ One of the most exciting features of C++11 is ability to create lambda functions (sometimes referred to as closures). What does this mean? A lambda function is a function that you can write inline in your source code (usually to pass in to another function, similar to the idea of a functor or function pointer). With lambda, creating quick functions has become much easier, and this means that not only can you start using lambda when you'd previously have needed to write a separate named function, but you can start writing more code that relies on the ability to create quick-and-easy functions. In this article, I'll first explain why lambda is great--with some examples--and then I'll walk through all of the details of what you can do with lambda.

Why Lambdas Rock

Imagine that you had an address book class, and you want to be able to provide a search function. You might provide a simple search function, taking a string and returning all addresses that match the string. Sometimes that's what users of the class will want. But what if they want to search only in the domain name or, more likely, only in the username and ignore results in the domain name? Or maybe they want to search for all email addresses that also show up in another list. There are a lot of potentially interesting things to search for. Instead of building all of these options into the class, wouldn't it be nice to provide a generic "find" method that takes a procedure for deciding if an email address is interesting? Let's call the method findMatchingAddresses, and have it take a "function" or "function-like" object.

```
#include <string>
#include <vector>

class AddressBook
{
    public:
    // using a template allows us to ignore the differences between functors, function pointers
    // and lambda
    template<typename Func>
    std::vector<std::string> findMatchingAddresses (Func func)
    {
        std::vector<std::string> results;
        for ( auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr )
        {
            // call the function passed into findMatchingAddresses and see if it matches
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }

    private:
    std::vector<std::string> _addresses;
};
```

Anyone can pass a function into the findMatchingAddresses that contains logic for finding a particular function. If the function returns true, when given a particular address, the address will be returned. This kind of approach was OK in earlier version of C++, but it suffered from one fatal flaw: it wasn't quite convenient enough to create functions. You had to go define it somewhere else, just to be able to pass it in for one simple use. That's where lambdas come in.

Basic Lambda Syntax

Before we write some code to solve this problem, let's see the really basic syntax for lambda.

Before we write some code to solve this problem, let's see the really basic syntax for lambda.

```
#include <iostream>

using namespace std;

int main()
{
    auto func = [] () { cout << "Hello world"; };
    func(); // now call the function
}
```

Okay, did you spot the lambda, starting with []? That identifier, called the capture specification, tells the compiler we're creating a lambda function. You'll see this (or a variant) at the start of every lambda function.

Next up, like any other function, we need an argument list: (). Where is the return value? Turns out that we don't need to give one. In C++11, if the compiler can deduce the return value of the lambda function, it will do it rather than force you to include it. In this case, the compiler knows the function returns nothing. Next we have the body, printing out "Hello World". This line of code doesn't actually cause anything to print out though--we're just creating the function here. It's almost like defining a normal function--it just happens to be inline with the rest of the code.

It's only on the next line that we call the lambda function: func() -- it looks just like calling any other function. By the way, notice how easy this is to do with auto! You don't need to sweat the ugly syntax of a function pointer.

Applying Lambda in our Example

Let's look at how we can apply this to our address book example, first creating a simple function that finds email addresses that contain ".org".

```
AddressBook global_address_book;

vector<string> findAddressesFromOrgs ()
{
    return global_address_book.findMatchingAddresses(
        // we're declaring a lambda here; the [] signals the start
        [] (const string& addr) { return addr.find( ".org" ) != string::npos; }
    );
}
```

Once again we start off with the capture specifier, [], but this time we have an argument--the address, and we check if it contains ".org". Once again, nothing inside the body of this lambda function is executed here yet; it's only inside findMatchingAddresses, when the variable func is used, that the code inside the lambda function executes.

In other words, each loop through findMatchingAddresses, it calls the lambda function and gives it the address as an argument, and the function checks if it contains ".org".

Variable Capture with Lambdas

Although these kinds of simple uses of lambd are nice, variable capture is the real secret sauce that makes a lambda function great. Let's imagine that you want to create a small function that finds addresses that contain a specific name. Wouldn't it be nice if you could write code something like this?

```
// read in the name from a user, which we want to search
string name;
cin>> name;
return global_address_book.findMatchingAddresses(
    // notice that the lambda function uses the the variable 'name'
    [&] (const string& addr) { return name.find( addr ) != string::npos; }
);
```

It turns out that this example is completely legal--and it shows the real value of lambda. We're able to take a variable declared outside of the lambda (name), and use it inside of the lambda. When findMatchingAddresses calls our lambda function, all the code inside of it executes--and when name.find is called, it has access to the name that the user passed in. The only thing we needed to do to make it work is tell the compiler we wanted to have variable capture. I did that by putting [&] for the capture specification, rather than []. The empty [] tells the compiler not to capture any variables, whereas the [&] specification tells the compiler to perform variable capture.

Isn't that marvelous? We can create a simple function to pass into the find method, capturing the variable name, and write it all in only a few lines of code. To get a similar behavior without C++11, we'd either need to create an entire functor class or we'd need a specialized method on AddressBook. In C++11, we can have a single simple interface to AddressBook that can support any kind of filtering really easily.

Just for fun, let's say that we want to find only email addresses that are longer than a certain number of characters. Again, we can do this easily:

```
int min_len = 0;
cin >> min_len;
return global_address_book.find( [&] (const string& addr) { return addr.length() >= min_len; } );
```

By the way, to steal a line from Herb Sutter, you should get used to seeing "});" This is the standard end-of-function-taking-lambda syntax, and the more you start seeing and using lambda in your own code, the more you'll see little piece of syntax.

Lambda and the STL

One of the biggest beneficiaries of lambda functions are, no doubt, power users of the standard template library algorithms package. Previously, using algorithms like for_each was an exercise in contortions. Now, though, you can use for_each and other STL algorithms almost as if you were writing a normal loop. Compare:

```
vector<int> v;
v.push_back( 1 );
v.push_back( 2 );
//...
for ( auto itr = v.begin(), end = v.end(); itr != end; itr++ )
{
    cout << *itr;
}
```

with

```
vector<int> v;
v.push_back( 1 );
v.push_back( 2 );
//...
for_each( v.begin(), v.end(), [] (int val)
{
    cout << val;
} );
```

That's pretty good looking code if you ask me--it reads, and is structured, like a normal loop, but you're suddenly able to take advantage of the goodness that for_each provides over a regular for loop--for example, guarantees that you have the right end condition. Now, you might wonder, won't this kill performance? Well, here's the kicker: it turns out that for_each is actually **faster** than a regular for loop. (The reason: it can take advantage of loop unrolling.)

If you're interested in more on C++11 lambda and the benefits to the STL, you'll enjoy this video of Herb Sutter talking about C++11 lambdas.

I hope this STL example shows you that lambda functions are more than just a slightly more convenient way of creating functions--they allow you to create entirely new ways of writing programs, where you have code that takes other functions as data and allows you to abstract away the proccessing of a particular data structure. for_each works on a list, but wouldn't it be great to have similar functions for working with trees, where all you had to do was right some code that would process each node, and not have to worry about the traversal algorithm? This kind of decomposition where one function worries about the structure of data, while delegating the data processing to another function can be quite powerful. With lambda, C++11 enables this new kind of programming. Not that you couldn't have done it before--for_each isn't new--it's just that you wouldn't have wanted to do it before.

More on the new Lambda Syntax

By the way, the parameter list, like the return value is also optional if you want a function that takes zero arguments. Perhaps the shortest possible lambda expression is:

```
[] {}
```

Which is a function that takes no arguments and does nothing. An only slightly more compelling example:

```
using namespace std;
#include <iostream>

int main()
{
    [] { cout << "Hello, my Greek friends"; }();
}
```

Personally, I'm not yet sold on omitting the argument list; I think the [] () structure tends to help lambda functions stand out a little more in the code, but time will tell what standards people come up with.

Return Values

By default, if your lambda does not have a return statement, it defaults to void. If you have a simple return expression, the compiler will deduce the type of the return value:

```
[] () { return 1; } // compiler knows this returns an integer
```

If you write a more complicated lambda function, with more than one return value, you should specify the return type. (Some compilers, like GCC, will let you get away without doing this, even if you have more than one return statement, but the standard doesn't guarantee it.)

Lambdas take advantage of the optional new C++11 return value syntax of putting the return value after the function. In fact, you must do this if you want to specify the return type of a lambda. Here's a more explicit version of the really simple example from above:

```
[] () -> int { return 1; } // now we're telling the compiler what we want
```

Throw Specifications

Although the C++ standards committee decided to deprecate throw specifications (except for a few cases I'll cover in a later article), they didn't remove them from the language, and there are tools that do static code analysis to check exception specifications, such as PC Lint. If you are using one of these tools to do compile time exception checking, you really want to be able to say which exceptions your lambda function throws. The main reason I can see for doing this is when you're passing a lambda function into another function as an argument, and that function expects the lambda function to throw only a specific set of exceptions. By providing an exception spec for your lambda function, you could allow a tool like PC Lint to check that for you. If you want to do that, it turns out you can. Here's a lambda that specifies that it takes no arguments and does not throw an exception:

```
[] () throw () { /* code that you don't expect to throw an exception*/ }
```

How are Lambda Closures Implemented?

So how does the magic of variable capture really work? It turns out that the way lambdas are implemented is by creating a small class; this class overloads the operator(), so that it acts just like a function. A lambda function is an instance of this class; when the class is constructed, any variables in the surrounding enviroment are passed into the constructor of the lambda function class and saved as member variables. This is, in fact, quite a bit like the idea of a functor that is already possible. The benefit of C++11 is that doing this becomes almost trivially easy--so you can use it all the time, rather than only in very rare circumstances where writing a whole new class makes sense.

C++, being very performance sensitive, actually gives you a ton of flexibility about what variables are captured, and how--all controlled via the capture specification, []. You've already seen two cases--with nothing in it, no variables are captured, and with &, variables are captured by reference. If you make a lambda with an empty capture group, [], rather than creating the class, C++ will create a regular function. Here's the full list of options:

[]	Capture nothing (or, a scorched earth strategy?)
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by making a copy
[=, &foo]	Capture any referenced variable by making a copy, but capture variable foo by reference
[bar]	Capture bar by making a copy; don't copy anything else
[this]	Capture the this pointer of the enclosing class

Notice the last capture option--you don't need to include it if you're already specifying a default capture (= or &), but the fact that you can capture the this pointer of a function is super-important because it means that you don't need to make a distinction between local variables and fields of a class when writing lambda functions. You can get access to both. The cool thing is that you don't need to explicitly use the this pointer; it's really like you are writing a function inline.

```
class Foo
{
public:
    Foo () : _x( 3 ) {}
    void func ()
    {
        // a very silly, but illustrative way of printing out the value of _x
        [this] () { cout << _x; } ();
    }

private:
        int _x;
};

int main()
{
    Foo f;
    f.func();
}
```

Dangers and Benefits of Capture by Reference

When you capture by reference, the lambda function is capable of modifying the local variable outside the lambda function--it is, after all, a reference. But this also means that if you return a lamba function from a function, you shouldn't use capture-by-reference because the reference will not be valid after the function returns.

What type is a Lambda?

The main reason that you'd want to create a lambda function is that someone has created a function that expects to receive a lambda function. We've already seen that we can use templates to take a lambda function as an argument, and auto to hold onto a lambda function as a local variable. But how do you name a specific lambda? Because each lambda function is implemented by creating a separate class, as you saw earlier, even single lambda function is really a different type--even if the two functions have the same arguments and the same return value! But C++11 does include a convenient wrapper for storing any kind of function--lambda function, functor, or function pointer: std::function.

std::function

The new std::function is a great way of passing around lambda functions both as parameters and as return values. It allows you to specify the exact types for the argument list and the return value in the template. Here's out AddressBook example, this time using std::function instead of templates. Notice that we do need to use the 'functional' header file.

```
#include <functional>
#include <vector>

class AddressBook
{
    public:
    std::vector<string> findMatchingAddresses (std::function<bool (const string&)> func)
    {
        std::vector<string> results;
        for ( auto itr = _addresses.begin(), end = _addresses.end(); itr != end; ++itr )
        {
            // call the function passed into findMatchingAddresses and see if it matches
            if ( func( *itr ) )
            {
                results.push_back( *itr );
            }
        }
        return results;
    }

    private:
    std::vector<string> _addresses;
};
```

One big advantage of std::function over templates is that if you write a template, you need to put the whole function in the header file, whereas std::function does not. This can really help if you're working on code that will change a lot and is included by many source files.

If you want to check if a variable of type std::function is currently holding a valid function, you can always treat it like a boolean:

```
std::function<int ()> func;
// check if we have a function (we don't since we didn't provide one)
if ( func )
{
    // if we did have a function, call it
    func();
}
```

A Note About Function Pointers

Under the final C++11 spec, if you have a lambda with an empty capture specification, then it can be treated like a regular function and assigned to a function pointer. Here's an example of using a function pointer with a capture-less lambda:

```
typedef int (*func)();
func = [] () -> { return 2; };

func();
```

This works because a lambda that doesn't have a capture group doesn't need its own class--it can be compiled to a regular old function, allowing it to be passed around just like a normal function. Unfortunately, support for this feature is not included in MSVC 10, as it was added to the standard too late.

Making Delegates with Lambdas

Let's look at one more example of a lambda function--this time to create a delegate. What's a delgate, you ask? When you call a normal function, all you need is the function itself. When you call a method on an object, you need two things: the function and the object itself. It's the difference between func() and obj.method(). To call a method, you need both. Just passing in the address of the method into a function isn't enough; you need to have an object to call the method on.

Let's look at an example, starting with some code that again expects a function as an argument, into which we'll pass a delegate.

```
#include <functional>
#include <string>

class EmailProcessor
{
public:
    void receiveMessage (const std::string& message)
    {
        if ( _handler_func )
        {
            _handler_func( message );
        }
        // other processing
    }
    void setHandlerFunc (std::function<void (const std::string&)> handler_func)
    {
        _handler_func = handler_func;
    }

private:
        std::function<void (const std::string&)> _handler_func;
};
```

This is a pretty standard pattern of allowing a callback function to be registered with a class when something interesting happens.

But now let's say we want another class that is responsible for keeping track of the longest message received so far (why do you want to do this? Maybe you are a bored sysadmin). Anyway, we might create a little class for this:

```
#include <string>

class MessageSizeStore
{
    MessageSizeStore () : _max_size( 0 ) {}
    void checkMessage (const std::string& message )
    {
        const int size = message.length();
        if ( size > _max_size )
        {
            _max_size = size;
        }
    }
    int getSize ()
    {
        return _max_size;
    }

private:
    int _max_size;
};
```

What if we want to have the method checkMessage called whenever a message arrives? We can't just pass in checkMessage itself--it's a method, so it needs an object.

```
EmailProcessor processor;
MessageSizeStore size_store;
processor.setHandlerFunc( checkMessage ); // this won't work
```

We need some way of binding the variable size_store into the function passed to setHandlerFunc. Hmm, sounds like a job for lambda!

```
EmailProcessor processor;
MessageSizeStore size_store;
MessageSizeStore size_store;
processor.setHandlerFunc(
        [&] (const std::string& message) { size_store.checkMessage( message ); }
);
```

Isn't that cool? We are using the lambda function here as glue code, allowing us to pass a regular function into setHandlerFunc, while still making a call onto a method--creating a simple delegate in C++.

In Summary

So are lambda functions really going to start showing up all over the place in C++ code when the language has survived for decades without them? I think so--I've started using lambda functions in production code, and they are starting to show up all over the place--in some cases shortening code, in some cases improving unit tests, and in some cases replacing what could previously have only been accomplished with macros. So yeah, I think lambdas rock way more than any other Greek letter.

Lambda functions are available in GCC 4.5 and later, as well as MSVC 10 and version 11 of the Intel compiler.

Next: Range-Based For Loops Range-based for loops make iterating over vectors and other containers very easy
Previous: How auto, decltype, and the new function syntax work together to create better code Learn about some of the new type inference features of C++11

C++11 range-based for loops

By Alex Allain

In the first article introducing C++11 I mentioned that C++11 will bring some nice usability improvements to the language. What I mean is that it removes unnecessary typing and other barriers to getting code written quickly. One example I've already covered is the new meaning of the auto keyword; now I'd like to talk more about the range-based for loop--both how to use it, and how to make your own classes work with it.

Basic syntax for range-based for loops

Nowadays, almost every programming language has a convenient way to write a for loop over a range of values. Finally, C++ has the same concept; you can provide a container to your for loop, and it will iterate over it. We've already seen a few basic examples in What is C++11? To refresh your memory, the range-based for loop looks like this:

```
vector<int> vec;
vec.push_back( 10 );
vec.push_back( 20 );

for (int i : vec )
{
    cout << i;
}
```

This code prints the contents of a vector called vec, with the variable i taking on the value of each element of the vector, in series, until the end of the vector is reached.

You can use auto in the type, to iterate over more complex data structures conveniently--for example, to iterate over a map you can write

```
map<string, string> address_book;
for ( auto address_entry : address_book )
{
            cout  << address_entry.first << " < " << address_entry.second << ">" << endl;
}
```

And you don't need to worry about spelling out the iterator type.

Modifying the Contents of the Vector

If you want to modify the values in the container you're looping over, or if you want to avoid copying large objects, and the underlying iterator supports it, you can make the loop variable a reference. For example, here's a loop that increments each element in an integer vector:

```
vector<int> vec;
vec.push_back( 1 );
vec.push_back( 2 );

for (int& i : vec )
{
    i++; // increments the value in the vector
}
for (int i : vec )
{
    // show that the values are updated
    cout << i << endl;
}
```

What does it mean to have a range?

Strings, arrays, and all STL containers can be iterated over with the new range-based for loop already. But what if you want to allow your own data structures to use the new syntax?

In order to make a data structure iterable, it must work similarly to the existing STL iterators.

- There must be begin and end methods that operate on that structure, either as members or as stand-alone functions, and that return iterators to the beginning and end of the structure
- The iterator itself must support an operator* method, an operator != method, and an operator++ method, either as members or as stand-alone functions (you can read more about operator overloading)

Note that operator++ should be the prefix version, which is done by declaring a function called operator++ that takes no arguments.

That's it! With these five functions, you can have a data structure that works with a range-based for loop. Because the begin and end methods can be non-member functions (begin(container) instead of container.begin()), you can even adapt existing data structures that don't natively support the STL-style iterators. All you must do is create your own iterator that supports *, prefix increment (++itr) and != and that has a way of defining a begin iterator and an end iterator.

Since range-based for loops are so nice, I suspect that most new containers that don't already support the STL iterator model will want to add adaptors of some sort that allow range-based for loops to be used. Here's a small program that demonstrates creating a simple iterator that works with a range-based for loops. In it, I create an IntVector type that is fixed at a size of 100, and that can be iterated over using a class called Iter. I've made this code const-correct, but if you haven't seen that concept before, the code works just fine with every const removed.

```
#include <iostream>

using namespace std;

// forward-declaration to allow use in Iter
class IntVector;

class Iter
{
    public:
    Iter (const IntVector* p_vec, int pos)
        : _pos( pos )
        , _p_vec( p_vec )
    { }

    // these three methods form the basis of an iterator for use with
    // a range-based for loop
    bool
    operator!= (const Iter& other) const
    {
        return _pos != other._pos;
    }

    // this method must be defined after the definition of IntVector
    // since it needs to use it
    int operator* () const;

    const Iter& operator++ ()
    {
        ++_pos;
        // although not strictly necessary for a range-based for loop
        // following the normal convention of returning a value from
        // operator++ is a good idea.
        return *this;
    }

    private:
    int _pos;
    const IntVector * _p_vec;
    const IntVector *_p_vec;
};

class IntVector
{
    public:
    IntVector ()
    {
    }

    int get (int col) const
    {
        return _data[ col ];
    }
    Iter begin () const
    {
        return Iter( this, 0 );
    }

    Iter end () const
    {
        return Iter( this, 100 );
    }

    void set (int index, int val)
    {
        _data[ index ] = val;
    }

    private:
   int _data[ 100 ];
};

int
Iter::operator* () const
{
     return _p_vec->get( _pos );
}

// sample usage of the range-based for loop on IntVector
int main()
{
    IntVector v;
    for ( int i = 0; i < 100; i++ )
    {
        v.set( i , i );
    }
    for ( int i : v ) { cout << i << endl; }
}
```

One thing to notice about this code is that it does not allow modification of elements in the IntVector by using a reference in the for loop. You should be able to add this easily by changing the return value of get to be a reference, but this would make the code much longer by requiring the addition of non-const methods, and I wanted to focus on the basic structure.

Do range-based for loops increase performance?

In my limited testing with GCC 4.6, I did not see a performance improvement in range-based for loops over normal STL iteration, but it seems like it should be possible for range-based for loops to see the same performance improvements as the STL's for_each function, which has the same model of iterating over a container in iterator order.

Compiler availability

Unfortunately, range-based for loops are not all that well supported. MSVC doesn't support them at all, and GCC only added support in version 4.6. In order to test out range-based for loops, I had to install GCC 4.6 from source; I found this article useful for doing so under Cygwin. You can also find pre-built MinGW binaries with GCC 4.7 on this page.

Next: Generalized Constant Expressions in C++11 Learn how C++11 makes compile-time processing easier than ever
Previous: Lambda Functions in C++11 - the definitive guide One of the most exciting features of C++11 is ability to create lambda functions, learn what they are and how to use them

Constexpr - Generalized Constant Expressions in C++11

By Alex Allain

There are several improvements in C++11 that promise to allow programs written using C++11 to run faster than ever before. One of those improvements, generalized constant expressions, allows programs to take advantage of compile-time computation. If you're familiar with template metaprogramming, constexpr will seem like a way of making your life much easier. If you're not familiar with template metaprogramming, that's ok--constexpr makes the benefits of compile-time programming much more widely accessible.

The basic idea of constant expressions is to allow certain computations to take place at compile time--literally while your code compiles--rather than when the program itself is run. This has an obvious performance benefit: if something can be done at compile time, it will be done once, rather than every time the program runs. Need to compute the value of a known constant like the sine or cosine of a specific, known-at-compile number? Sure, you could use the library sin or cos function, but you'll pay the price of a function call at runtime. With constexpr, you can create a function that, at compile time, computes the value for you. Your user's CPU will never need to do any work at all for it!

Now, it's certainly true that if the operation can be computed at compile time, you could hard-code a constant. But doing that means that you lose flexibility if you later want to change the constant value--you have to recompute it, rather than just change an argument to a function.

Constexpr in action

In order to get compile time processing, you need to use the constext keyword on the function that you want to be able to compute at compile time.

```
constexpr int multiply (int x, int y)
{
    return x * y;
}

// the compiler may evaluate this at compile time
const int val = multiply( 10, 10 );
```

Another benefit of constexpr, beyond the performance of compile time computation, is that it allows functions to be used in all sorts of situations that previously would have called for macros. For example, let's say you want to have a function that computes the the size of an array based on some multiplier. If you had wanted to do this in C++ without a constexpr, you'd have needed to create a macro or used template metaprogramming since you can't use the result of a function call to declare an array. But with constexpr, you can now use a call to a constexpr function inside an array declaration:

```
constexpr int getDefaultArraySize (int multiplier)
{
    return 10 * multiplier;
}

int my_array[ getDefaultArraySize( 3 ) ];
```

Restrictions on constexpr functions

A constexpr function has some very rigid rules it must follow:

- It must consist of single return statement (with a few exceptions)
- It can call only other constexpr functions
- It can reference only constexpr global variables

Notice that one thing that isn't restricted is recursion. How can you do recursion if the function can only have a single return statement? By using the ternary operator (sometimes known as the question mark colon operator). For example, here's a function that computes the value of a specific factorial:

```
constexpr factorial (int n)
{
    return n > 0 ? n * factorial( n - 1 ) : 1;
}
```

Now you can use factorial(2) and when the compiler sees it, it can optimize away the call and make the calculation entirely at runtime. In this way, by allowing more sophisticated calculations, constexpr behaves differently than a mere inline function. You can't inline a recursive function! In fact, any time the function argument is itself a constexpr, it can be computed at compile time.

What else can go in a constexpr function?

A constexpr function can have only a single line of executable code, but it may contain typedefs, using declarations and directives, and static_asserts.

Constexpr and runtime

A function declared as constexpr can also be called at runtime if the argument to the function is a non-constant--for example:

```
int n;
cin >> n;
factorial( n );
```

This means that you do not need to create separate functions for compile time and run time.

Using objects at compile time

Since any object that is referenced by a constexpr function must be constexpr, what if you want to use an object in that function? For example, what if you had a circle object?

```
class Circle
{
    public:
    Circle (int x, int y, int radius) : _x( x ), _y( y ), _radius( radius ) {}
    double getArea () const
    {
        return _radius * _radius * 3.1415926;
    }
    private:
        int _x;
        int _y;
        int _radius;
};
```

And you wanted to construct a circle at compile time and get its area?

```
constexpr Circle c( 0, 0, 10 );
constexpr double area = c.getArea();
```

It turns out that you can do this with a few small modifications to the Circle class. First, we need to declare the constructor as constexpr, and second, we need to declare the getArea function as constexpr. Declaring the constructor as a constexpr allows it to be run at compile time as long as it consists only of member initializations using other constexpr constructors (a compiler-generated default constructor can also be treated as constexpr, assuming all its members have constructors that are constexpr). Declaring the method getArea as constexpr allows it to be called at compile time:

```
class Circle
{
    public:
    constexpr Circle (int x, int y, int radius) : _x( x ), _y( y ), _radius( radius ) {}
    constexpr double getArea ()
    {
        return _radius * _radius * 3.1415926;
    }
    private:
        int _x;
        int _y;
        int _radius;
};
```

constexpr vs const

If you declare a class member function to be constexpr, that marks the function as 'const' as well. (Clearly it must be const if it is constexpr, because a constexpr function cannot modify the object in any way.) If you declare a variable as constexpr, that in turn marks the variable as const. However, it doesn't work the other way--a const function is not a constexpr, nor is a const variable a constexpr.

Constexpr and Floating Point Numbers

So far everything we've seen with constexpr could have been achieved--much more verbosely--using template metaprogramming. There is, however, one completely new piece of functionality enabled const constexpr: compile time computation of floating point values. Because double and float are not valid template parameter types, you can't easily use template metaprogramming to compute these values at compile time (for example, computing arbitrary values of sine and cosine). You'd have to fall back to using fixed point arithmetic. On the other hand, constexpr does allow the use of floating point values. Think about the possibility of compile time computation of values that are likely to show up in game or graphics programming. For example, the trigonometric functions sine and cosine are often used for computing object rotation (among other things). You can use the Taylor series for sine to compute sine values at compile time for constant arguments to sine.

Tradeoffs of constexpr

C++ already suffers from relatively slow compilation due to the need to recompile any code after changing a header file. Constexpr is sufficiently powerful that it risks introducing additional compile-time overhead. However, there are some built-in advantage sto constexpr that limit this risk. First, because constexpr functions always return the same output value for the same input, they can be memoized, and in fact GCC already supports memoization.

Due to the ability to memoize constexpr functions, in cases where constant expressions replace template metaprogramming, the performance impact shouldn't be worse than today, while the code will be much clearer. In fact, by eliminating the need to do a large number of template instantiations, the compilation time may actually be much faster.

Finally, the standard also allows compilers to limit the levels of nesting allowed for recursive constexpr methods (although it does require at least 512 levels to be allowed). This limitation may limit performance penalties for extremely high-overhead compile time calculations by preventing too much reliance on deep recursion. It's also useful to know about in case you do plan to write highly nested calculations.

Constexpr compiler availability

Constexpr requires the compiler to be able to do recursive function call processing at compile time, so it may not be a surprise that C++ compiler support for constexpr is pretty limited. In fact, the only compiler version I know of that fully supports constexpr is GCC 4.7. (Earlier versions of GCC allowed the syntax, but did not provide compile time processing.) If you want to try it out, you can either build GCC from source or pick up a pre-built Cygwin-compatible binary at this page.

Next: Faster Code with Rvalue References and Move Semantics Learn how C++11 lets you write faster code in yet another way, by avoiding slow copies in favor of fast moves
Previous: Range-Based For Loops Range-based for loops make iterating over vectors and other containers very easy

Move semantics and rvalue references in C++11

By Alex Allain

C++ has always produced fast programs. Unfortunately, until C++11, there has been an obstinate wart that slows down many C++ programs: the creation of temporary objects. Sometimes these temporary objects can be optimized away by the compiler (the return value optimization, for example). But this is not always the case, and it can result in expensive object copies. What do I mean?

Let's say that you have the following code:

```
#include <iostream>

using namespace std;

vector<int> doubleValues (const vector<int>& v)
{
    vector<int> new_values( v.size() );
    for (auto itr = new_values.begin(), end_itr = new_values.end(); itr != end_itr; ++itr )
    {
        new_values.push_back( 2 * *itr );
    }
    return new_values;
}

int main()
{
    vector<int> v;
    for ( int i = 0; i < 100; i++ )
    {
        v.push_back( i );
    }
    v = doubleValues( v );
}
```

If you've done a lot of high performance work in C++, sorry about the pain that brought on. If you haven't--well, let's walk through why this code is terrible C++03 code. (The rest of this tutorial will be about why it's fine C++11 code.) The problem is with the copies. When doubleValues is called, it constructs a vector, new_values, and fills it up. This alone might not be ideal performance, but if we want to keep our original vector unsullied, we need a second copy. But what happens when we hit the return statement?

The entire contents of new_values must be copied! In principle, there could be up to two copies here: one into a temporary object to be returned, and a second when the vector assignment operator runs on the line v = doubleValues(v);. The first copy may be optimized away by the compiler automatically, but there is no avoiding that the assignment to v will have to copy all the values again, which requires a new memory allocation and another iteration over the entire vector.

This example might be a little bit contrived--and of course you can find ways to avoid this kind of problem--for example, by storing and returning the vector by pointer, or by passing in a vector to be filled up. The thing is, neither of these programming styles is particularly natural. Moreover, an approach that requires returning a pointer has introduced at least one more memory allocation, and one of the design goals of C++ is to avoid memory allocations.

The worst part of this whole story is that the object returned from doubleValues is a temporary value that's no longer needed. When you have the line v = doubleValues(v), the result of doubleValues(v) is just going to get thrown away once it is copied! In theory, it should be possible to skip the whole copy and just pilfer the pointer inside the temporary vector and keep it in v. In effect, why can't we **move** the object? In C++03, the answer is that there was no way to tell if an object was a temporary or not, you had to run the same code in the assignment operator or copy constructor, no matter where the value came from, so no pilfering was possible. In C++11, the answer is--you can!

That's what rvalue references and move semantics are for! Move semantics allows you to avoid unnecessary copies when working with temporary objects that are about to evaporate, and whose resources can safely be taken from that temporary object and used by another.

Move semantics relies on a new feature of C++11, called rvalue references, which you'll want to understand to really appreciate what's going on. So first let's talk about what an rvalue is, and then what an rvalue reference then. Finally, we'll come back to move semantics and how it can be implemented with rvalue references.

Rvalues and lvalues - bitter rivals, or best of friends?

In C++, there are rvalues and lvalues. An lvalue is an expression whose address can be taken, a locator value--essentially, an lvalue provides a (semi)permanent piece of memory. You can make assignments to lvalues. For example:

```
int a;
a = 1; // here, a is an lvalue
```

You can also have lvalues that aren't variables:

```
int x;
int& getRef ()
{
        return x;
}

getRef() = 4;
```

Here, getRef returns a reference to a global variable, so it's returning a value that is stored in a permanent location. (You could literally write & getRef() if you wanted to, and it would give you the address of x.)

Rvalues are--well, rvalues are not lvalues. An expression is an rvalue if it results in a temporary object. For example:

```
int x;
int getVal ()
{
    return x;
}
getVal();
```

Here, getVal() is an rvalue--the value being returned is not a reference to x, it's just a temporary value. This gets a little bit more interesting if we use real objects instead of numbers:

```
string getName ()
{
    return "Alex";
}
getName();
```

Here, getName returns a string that is constructed inside the function. You can assign the result of getName to a variable:

```
string name = getName();
```

But you're assigning from a temporary object, not from some value that has a fixed location. getName() is an rvalue.

Detecting temporary objects with rvalue references

The important thing is that rvalues refer to temporary objects--just like the value returned from doubleValues. Wouldn't it be great if we could know, without a shadow of a doubt, that a value returned from an expression was a temporary, and somehow write code that is overloaded to behave differently for temporary objects? Why, yes, yes indeed it would be. And this is what rvalue references are for. An rvalue reference is a reference that will bind only to a temporary object. What do I mean?

Prior to C++11, if you had a temporary object, you could use a "regular" or "lvalue reference" to bind it, but only if it was const:

```
const string& name = getName(); // ok
string& name = getName(); // NOT ok
```

The intuition here is that you cannot use a "mutable" reference because, if you did, you'd be able to modify some object that is about to disappear, and that would be dangerous. Notice, by the way, that holding on to a const reference to a temporary object ensures that the

temporary object isn't immediately destructed. This is a nice guarantee of C++, but it is still a temporary object, so you don't want to modify it.

In C++11, however, there's a new kind of reference, an "rvalue reference", that will let you bind a mutable reference to an rvalue, but not an lvalue. In other words, rvalue references are perfect for detecting if a value is temporary object or not. Rvalue references use the && syntax instead of just &, and can be const and non-const, just like lvalue references, although you'll rarely see a const rvalue reference (as we'll see, mutable references are kind of the point):

```
const string&& name = getName(); // ok
string&& name = getName(); // also ok - praise be!
```

So far this is all well and good, but how does it help? The most important thing about lvalue references vs rvalue references is what happens when you write functions that take lvalue or rvalue references as arguments. Let's say we have two functions:

```
printReference (const String& str)
{
        cout << str;
}

printReference (String&& str)
{
        cout << str;
}
```

Now the behavior gets interesting--the printReference function taking a const lvalue reference will accept any argument that it's given, whether it be an lvalue or an rvalue, and regardless of whether the lvalue or rvalue is mutable or not. However, in the presence of the second overload, printReference taking an rvalue reference, it will be given all values *except* mutable rvalue-references. In other words, if you write:

```
string me( "alex" );
printReference(  me ); // calls the first printReference function, taking an lvalue reference

printReference( getName() ); // calls the second printReference function, taking a mutable rvalue reference
```

Now we have a way to determine if a reference variable refers to a temporary object or to a permanent object. The rvalue reference version of the method is like the secret back door entrance to the club that you can only get into if you're a temporary object (boring club, I guess). Now that we have our method of determining if an object was a temporary or a permanent thing, how can we use it?

Move constructor and move assignment operator

The most common pattern you'll see when working with rvalue references is to create a move constructor and move assignment operator (which follows the same principles). A move constructor, like a copy constructor, takes an instance of an object as its argument and creates a new instance based on the original object. However, the move constructor can avoid memory reallocation because we know it has been provided a temporary object, so rather than copy the fields of the object, we will move them.

What does it mean to move a field of the object? If the field is a primitive type, like int, we just copy it. It gets more interesting if the field is a pointer: here, rather than allocate and initialize new memory, we can simply steal the pointer and null out the pointer in the temporary object! We know the temporary object will no longer be needed, so we can take its pointer out from under it.

Imagine that we have a simple ArrayWrapper class, like this:

```
class ArrayWrapper
{
    public:
        ArrayWrapper (int n)
            : _p_vals( new int[ n ] )
            , _size( n )
        {}
        // copy constructor
        ArrayWrapper (const ArrayWrapper& other)
            : _p_vals( new int[ other._size  ] )
            , _size( other._size )
        {
            for ( int i = 0; i < _size; ++i )
            {
                _p_vals[ i ] = other._p_vals[ i ];
            }
        }
        ~ArrayWrapper ()
        {
            delete [] _p_vals;
        }
    private:
    int *_p_vals;
    int _size;
};
```

Notice that the copy constructor has to both allocate memory and copy every value from the array, one at a time! That's a lot of work for a copy. Let's add a move constructor and gain some massive efficiency.

```
class ArrayWrapper
{
public:
    // default constructor produces a moderately sized array
    ArrayWrapper ()
        : _p_vals( new int[ 64 ] )
        , _size( 64 )
    {}

    ArrayWrapper (int n)
        : _p_vals( new int[ n ] )
        , _size( n )
    {}

    // move constructor
    ArrayWrapper (ArrayWrapper&& other)
        : _p_vals( other._p_vals  )
        , _size( other._size )
    {
        other._p_vals = NULL;
    }

    // copy constructor
    ArrayWrapper (const ArrayWrapper& other)
        : _p_vals( new int[ other._size  ] )
        , _size( other._size )
    {
        for ( int i = 0; i < _size; ++i )
        {
            _p_vals[ i ] = other._p_vals[ i ];
        }
    }
    ~ArrayWrapper ()
    {
        delete [] _p_vals;
    }

private:
    int *_p_vals;
    int _size;
};
```

Wow, the move constructor is actually simpler than the copy constructor! That's quite a feat. The main things to notice are:

1. The parameter is a non-const rvalue reference
2. other._p_vals is set to NULL

The second observation explains the first--we couldn't set other._p_vals to NULL if we'd taken a const rvalue reference. But why do we need to set other._p_vals = NULL? The reason is the destructor--when the temporary object goes out of scope, just like all other C++ objects, its destructor will run. When its destructor runs, it will free _p_vals. The same _p_vals that we just copied! If we don't set other._p_vals to NULL, the move would not really be a move--it would just be a copy that introduces a crash later on once we start using freed memory. This is the whole point of a move constructor: to avoid a copy by changing the original, temporary object!

Again, the overload rules work such that the move constructor is called only for a temporary object--and only a temporary object that can be modified. One thing this means is that if you have a function that returns a const object, it will cause the copy constructor to run instead of the move constructor--so don't write code like this:

```
const ArrayWrapper getArrayWrapper (); // makes the move constructor useless, the temporary is const!
```

There's still one more situation we haven't discussed how to handle in a move constructor--when we have a field that is an object. For example, imagine that instead of having a size field, we had a metadata field that looked like this:

```
class MetaData
{
public:
    MetaData (int size, const std::string& name)
        : _name( name )
        , _size( size )
    {}

    // copy constructor
    MetaData (const MetaData& other)
        : _name( other._name )
        , _size( other._size )
    {}

    // move constructor
    MetaData (MetaData&& other)
        : _name( other._name )
        , _size( other._size )
    {}

    std::string getName () const { return _name; }
    int getSize () const { return _size; }
    private:
    std::string _name;
    int _size;
};
```

Now our array can have a name and a size, so we might have to change the definition of ArrayWrapper like so:

```cpp
class ArrayWrapper
{
public:
    // default constructor produces a moderately sized array
    ArrayWrapper ()
        : _p_vals( new int[ 64 ] )
        , _metadata( 64, "ArrayWrapper" )
    {}

    ArrayWrapper (int n)
        : _p_vals( new int[ n ] )
        , _metadata( n, "ArrayWrapper" )
    {}

    // move constructor
    ArrayWrapper (ArrayWrapper&& other)
        : _p_vals( other._p_vals  )
        , _metadata( other._metadata )
    {
        other._p_vals = NULL;
    }

    // copy constructor
    ArrayWrapper (const ArrayWrapper& other)
        : _p_vals( new int[ other._metadata.getSize() ] )
        , _metadata( other._metadata )
    {
        for ( int i = 0; i < _metadata.getSize(); ++i )
        {
            _p_vals[ i ] = other._p_vals[ i ];
        }
    }
    ~ArrayWrapper ()
    {
        delete [] _p_vals;
    }
private:
    int *_p_vals;
    MetaData _metadata;
};
```

Does this work? It seems very natural, doesn't it, to just call the MetaData move constructor from within the move constructor for ArrayWrapper? The problem is that this just doesn't work. The reason is simple: the value of other in the move constructor--it's an rvalue reference. But an rvalue reference is not, in fact, an rvalue. It's an lvalue, and so the copy constructor is called, not the move constructor. This is weird. I know--it's confusing. Here's the way to think about it. A rvalue is an expression that creates an object that is about to evaporate into thin air. It's on its last legs in life--or about to fulfill its life purpose. Suddenly we pass the temporary to a move constructor, and it takes on new life in the new scope. In the context where the rvalue expression was evaluated, the temporary object really is over and done with. But in our constructor, the object has a name; it will be alive for the entire duration of our function. In other words, we might use the variable other more than once in the function, and the temporary object has a defined location that truly persists for the entire function. It's an lvalue in the true sense of the term locator value, we can locate the object at a particular address that is stable for the entire duration of the function call. We might, in fact, want to use it later in the function. If a move constructor were called whenever we held an object in an rvalue reference, we might use a moved object, by accident!

```
// move constructor
ArrayWrapper (ArrayWrapper&& other)
    : _p_vals( other._p_vals  )
    , _metadata( other._metadata )
{
    // if _metadata( other._metadata ) calls the move constructor, using
    // other._metadata here would be extremely dangerous!
    other._p_vals = NULL;
}
```

Put a final way: both lvalue and rvalue references are lvalue expressions. The difference is that an lvalue reference must be const to hold a reference to an rvalue, whereas an rvalue reference can always hold a reference to an rvalue. It's like the difference between a pointer, and what is pointed to. The thing pointed-to came from an rvalue, but when we use rvalue reference itself, it results in an lvalue.

std::move

So what's the trick to handling this case? We need to use std::move, from <utility>--std::move is a way of saying, "ok, honest to God I know I have an lvalue, but I want it to be an rvalue." std::move does not, in and of itself, move anything; it just turns an lvalue into an rvalue, so that you can invoke the move constructor. Our code should look like this:

```
#include <utilityh> // for std::move

    // move constructor
    ArrayWrapper (ArrayWrapper&& other)
        : _p_vals( other._p_vals  )
        , _metadata( std::move( other._metadata ) )
    {
        other._p_vals = NULL;
    }
```

And of course we should really go back to MetaData and fix its own move constructor so that it uses std::move on the string it holds:

```
MetaData (MetaData&& other)
    : _name( std::move( other._name ) ) // oh, blissful efficiency
    : _size( other._size )
{}
```

Move assignment operator

Just as we have a move constructor, we should also have a move assignment operator. You can easily write one using the same techniques as for creating a move constructor.

Move constructors and implicitly generated constructors

As you know, in C++ when you declare any constructor, the compiler will no longer generate the default constructor for you. The same is true here: adding a move constructor to a class will require you to declare and define your own default constructor. On the other hand, declaring a move constructor does not prevent the compiler from providing an implicitly generated copy constructor, and declaring a move assignment operator does not inhibit the creation of a standard assignment operator.

How does std::move work

You might be wondering, how does one write a function like std::move? How do you get this magical property of transforming an rvalue reference into an lvalue reference? The answer, as you might guess, is typecasting. The actual declaration for std::move is somewhat more involved, but at its heart, it's just a static_cast to an rvalue reference. This means, actually, that you don't really *need* to use move--but you should, since it's much more clear what you mean. The fact that a cast is required is, by the way, a very good thing! It means that you cannot accidentally convert an lvalue into an rvalue, which would be dangerous since it might allow an accidental move to take place. You must explicitly use std::move (or a cast) to convert an lvalue into an rvalue reference, and an rvalue reference will never bind to an lvalue on its own.

Returning an explicit rvalue-reference from a function

Are there ever times where you should write a function that returns an rvalue reference? What does it mean to return an rvalue reference anyway? Aren't functions that return objects by value already rvalues?

Let's answer the second question first: returning an explicit rvalue reference is different than returning an object by value. Take the following simple example:

```
int x;

int getInt ()
{
    return x;
}

int && getRvalueInt ()
{
    // notice that it's fine to move a primitive type--remember, std::move is just a cast
    return std::move( x );
}
```

Clearly in the first case, despite the fact that getInt() is an rvalue, there is a copy of the variable x being made. We can even see this by writing a little helper function:

```
void printAddress (const int& v) // const ref to allow binding to rvalues
{
    cout << reinterpret_cast<const void*>( & v ) << endl;
}

printAddress( getInt() );
printAddress( x );
```

When you run this program, you'll see that there are two separate values printed.

On the other hand,

```
printAddress( getRvalueInt() );
printAddress( x );
```

prints the same value because we are explicitly returning an rvalue here.

So returning an rvalue reference is a different thing than not returning an rvalue reference, but this difference manifests itself most noticeably if you have a pre-existing object you are returning instead of a temporary object created in the function (where the compiler is likely to eliminate the copy for you).

Now on to the question of whether you want to do this. The answer is: probably not. In most cases, it just makes it more likely that you'll end up with a dangling reference (a case where the reference exists, but the temporary object that it refers to has been destroyed). The issue is quite similar to the danger of returning an lvalue reference--the referred-to object may no longer exist. Rvalue references cannot magically keep an object alive for you. Returning an rvalue reference would primarily make sense in very rare cases where you have a member function and need to return the result of calling std::move on a field of the class from that function--and how often are you going to do that?

Move semantics and the standard library

Going back to our original example--we were using a vector, and we don't have control over the vector class and whether or not it has a move constructor or move assignment operator. Fortunately, the standards committee is wise, and move semantics has been added to the standard library. This means that you can now efficiently return vectors, maps, strings and whatever other standard library objects you want, taking full advantage of move semantics.

Moveable objects in STL containers

In fact, the standard library goes one step further. If you enable move semantics in your own objects by creating move assignment operators and move constructors, when you store those objects in a container, the STL will automatically use std::move, automatically taking advantage of move-enabled classes to eliminate inefficient copies.

Move semantics and rvalue reference compiler support

Rvalue references are supported by GCC, the Intel compiler and MSVC.

Previous: Generalized Constant Expressions in C++11 Learn how C++11 makes compile-time processing easier than ever
Next: Nullptr and Strongly Typed Enums (Enum classes) in C++11 Learn how C++11 improves type safety

Better types in C++11 - nullptr, enum classes (strongly typed enumerations) and cstdint

By Alex Allain

C++ has from the beginning attempted to improve on the type system of C, adding features like classes that let you build better types and enums, which eliminate the need for some uses of the preprocessor (which is not at all type safe). C++ also performs fewer implicit type conversions for you (such as not allowing implicit assignment from void*), letting the compiler find more bugs for you.

C++11 goes even further. Even though enums got rid of the need for integer #define constants, we still had the ugly, poorly typed NULL pointer. C++11 cleans this up by adding an explicit, clear nullptr value with its own type. C++11 also brings new, strongly typed enums. In this article, I'll cover both these improvements.

Why do we need strongly typed enums?

So why do we need strongly typed enums anyway? Old-style C++ enums are essentially integers; they could be compared with integers or with other enums of different types. The thing is, you normally don't want to do that since enums are supposed to be some fixed list of enumerated values. Why would you want to compare to some other enum type (or an integer)? It's like saying, "please compare this kind of nail with this kind of toothbrush." It makes no sense, and you probably don't mean to do it. But old-style C++ enums will happily tell you, "why yes, this nail isn't like this toothbrush" or, worse, they might compare equal because they happen to share the same underlying integer value ("ah yes, this nail IS a Panasonic electronic toothbrush"). Now, with strongly typed enums, the compiler will tell you that you're doing it. If you really mean it, you can always use a typecast.

Another limitation is that enum values were unscoped--in other words, you couldn't have two enumerations that shared the same name:

```
// this code won't compile!
enum Color {RED, GREEN, BLUE};
enum Feelings {EXCITED, MOODY, BLUE};
```

Strongly typed enums - enum classes

Enter strongly typed enums--and I don't mean **enums**. Strongly typed enums are a new kind of enum, declared like so:

```
// this code will compile (if your compiler supports C++11 strongly typed enums)
enum class Color {RED, GREEN, BLUE};
enum class Feelings {EXCITED, MOODY, BLUE};
```

The use of the word class is meant to indicate that each enum type really is different and not comparable to other enum types. Strongly typed enums, enum classes, also have better scoping. Each enum value is scoped within the name of the enum class. In other words, to access the enum values, you must write:

```
Color color = Color::GREEN;
if ( Color::RED == color )
{
    // the color is red
}
```

Old style C++ enums are still available--if you want them--largely for backward compatibility with existing code bases. They did pick up one trick; you can now, optionally, put the enum name in front of the value: Color::RED. But since this is optional, it doesn't solve any naming conflicts; it just makes it a little bit more clear.

Enum classes have another advantages over old-style enums. You can have a forward declaration to a strongly typed enum, meaning that you can write code like:

```
enum class Mood;

void assessMood (Mood m);

// later on:
enum class Mood { EXCITED, MOODY, BLUE };
```

Why would this be useful? Forward declarations are often about the physical layout of code on disk into different files or to provide opaque objects as part of an API. In the first case, where you care about the physical disk layout, using a forward declaration allows you to declare an enum type in the header file while putting specific values into the cpp file. This lets you change the list of possible enum values quite frequently without forcing all dependent files to recompile. In the second case, an enum class can be exposed as a type-safe but otherwise opaque value returned from one API function to be passed into another API function. The code using the API need not know the possible values the type can take on. Since the compiler still knows about the type, it can enforce that variables declared to work with that type are not confused with variables working with another type.

Well-defined enum sizes

A final advantage of enum classes is that you can set the size of your enum--you can use any signed or unsigned integer type. It defaults to int, but you can also use char, unsigned long, etc. This will ensure some measure of compatibility across compilers.

```
// we only have three colors, so no need for ints!
enum class Colors : char { RED = 1, GREEN = 2, BLUE = 3 };
```

But in C++11, we can do even better, specifying exact sizes for enums, using cstdint.

<cstdint>

One problem that C++ has suffered from is a lack of standard types that provide fixed, well-defined sizes. For example, sometimes you want to have a 32-bit integer, not just an int that might have different sizes on different architectures. In C++11, the C99 header file stdint.h has been included as cstdint. The cstdint header includes types such as std::int8_t, std::int16_t, std::int32_t, and std::int64_t (as well as unsigned versions that begin with u: std::uint8_t).

Here's an example that combines these new types with enum classes to get completely known sizes for your enum across compilers and architectures:

```
#include <cstdint>
enum class Colors : std::int8_t { RED = 1, GREEN = 2, BLUE = 3 };
```

nullptr

In C and C++, it's always been important to express the idea of a NULL pointer--one that has no value. Oddly, in C++, the expression used, 0 (or NULL, always #defined to zero) was not even a pointer type. Although this worked most of the time, it could lead to strange and unexpected problems in what are, admittedly, rather edge cases. For example imagine you have the following two function declarations:

```
void func(int n);
void func(char *s);

func( NULL ); // guess which function gets called?
```

Although it looks like the second function will be called--you are, after all, passing in what seems to be a pointer--it's really the first function that will be called! The trouble is that because NULL is 0, and 0 is an integer, the first version of func will be called instead. This is the kind of thing that, yes, doesn't happen all the time, but when it does happen, is extremely frustrating and confusing. If you didn't know the details of what is going on, it might well look like a compiler bug. A language feature that looks like a compiler bug is, well, not something you want.

Enter nullptr. In C++11, nullptr is a new keyword that can (and should!) be used to represent NULL pointers; in other words, wherever you were writing NULL before, you should use nullptr instead. It's no more clear to you, the programmer, (everyone knows what NULL means), but it's more explicit to the compiler, which will no longer see 0s everywhere being used to have special meaning when used as a pointer.

std::nullptr_t

nullptr, by the way, is not only declared to be a pointer and convert implicitly to all pointer types (and bool), but it is its own special, distinct type:

```
decltype( nullptr )
```

While we can use decltype to extract its type, there is also a more convenient notation:

```
std::nullptr_t
```

Since nullptr is its own unique type, you can use it as a constructor or function argument when you want to be sure that you only ever take an empty pointer for a value. For example:

```
void func( std::nullptr_t );
```

declares a function that takes only nullptr (or a value cast to std::nullptr_t) and nothing else, a rather neat trick.

Regardless of all this--the rule of thumb for C++11 is simply to start using nullptr whenever you would have otherwise used NULL in the past.

Previous: Faster Code with Rvalue References and Move Semantics Learn how C++11 lets you write faster code in yet another way, by avoiding slow copies in favor of fast moves