Rvalue References: C++0x Features in VC10, Part 2

Visual CPP Team

February 3rd, 2009 | 1 | 0

Part 1 of this series covered **lambdas**, **auto**, and **static_assert**.

Today, I'm going to talk about **rvalue references**, which enable two different things: **move semantics** and **perfect forwarding**. This post will be long, because I'm going to explain how rvalue references work in great detail. They're initially very confusing because they distinguish lvalues from rvalues, which very few C++98/03 programmers are extensively familiar with.

Fear not, for using rvalue references is easy, much easier than it initially sounds. Implementing either move semantics or perfect forwarding in your own code boils down to following simple patterns, which I will demonstrate. And it's definitely worth learning how to use rvalue references, as move semantics can produce *order of magnitude* performance improvements, and perfect forwarding makes writing highly generic code very easy.

lvalues and rvalues in C++98/03

In order to understand rvalue references in C++0x, you must first understand lvalues and rvalues in C++98/03.

The terminology of "lvalues" and "rvalues" is confusing because their history is confusing. (By the way, they're just pronounced as "L values" and "R values", although they're written as single words.) These concepts originally came from C, and then were elaborated upon by C++. To save time, I'll skip over their history, including why they're called "lvalues" and "rvalues", and I'll go directly to how they work in C++98/03. (Okay, it's not a big secret: "L" stands for "left" and "R" stands for "right". But the concepts have evolved since the names were chosen, and the names aren't very accurate anymore. Instead of going through the whole history lesson, you can consider the names to be arbitrary like "up quark" and "down quark", and you won't lose anything.)

C++03 3.10/1 says: "Every expression is either an *lvalue* or an *rvalue*." It's important to remember that lvalueness versus rvalueness is a property of expressions, not of objects.

Lvalues name objects that persist beyond a single expression. For example, `obj` , `*ptr` , `ptr[index]` , and `++x` are all lvalues.

Rvalues are temporaries that evaporate at the end of the full-expression in which they live ("at the semicolon"). For example, `1729` , `x + y` , `std::string("meow")` , and `x++` are all rvalues.

Notice the difference between `++x` and `x++` . If we have `int x = 0;` then the expression `x` is an lvalue, as it names a persistent object. The expression `++x` is also an lvalue. It modifies and then names the persistent object. However, the expression `x++` is an rvalue. It copies the original value of the persistent object, modifies the persistent object, and then returns the copy. This copy is a temporary.

Both ++x and x++ increment x, but ++x returns the persistent object itself, while x++ returns a temporary copy. That's why ++x is an lvalue, while x++ is an rvalue. Lvalueness versus rvalueness doesn't care about what an expression *does*, it cares about what an expression *names* (something persistent or something temporary).

If you want to build up intuition for this, another way to determine whether an expression is an lvalue is to ask "can I take its address?". If you can, it's an lvalue. If you can't, it's an rvalue. For example, &obj , &*ptr , &ptr[index] , and &++x are all valid (even though some of those expressions are silly), while &1729 , &(x + y) , &std::string("meow") , and &x++ are all invalid. Why does this work?&nbs p; The address-of operator requires that its "operand shall be an lvalue" (C++03 5.3.1/2). Why does it require that? Taking the address of a persistent object is fine, but taking the address of a temporary would be extremely dangerous, because temporaries evaporate quickly.

The preceding examples ignore operator overloading, which is convenient syntax for a function call. "A function call is an lvalue if and only if the result type is a reference." (C++03 5.2.2/10) Therefore, given vector<int> v(10, 1729); , v[0] is an lvalue because operator[]() returns int& (and &v[0] is valid and useful), while given string s("foo"); and string t("bar"); , s + t is an rvalue because operator+() returns string (and &(s + t) is invalid).

Both lvalues and rvalues can be either modifiable (non-const) or non-modifiable (const). Here are examples:

```
string one("cute");

const string two("fluffy");

string three() { return "kittens"; }

const string four() { return "are an essential part of a healthy diet"; }
```

```
one;     // modifiable lvalue

two;     // const lvalue

three(); // modifiable rvalue

four();  // const rvalue
```

Type& binds to modifiable lvalues (and can be used to observe and mutate them). It can't bind to const lvalues, as that would violate const correctness. It can't bind to modifiable rvalues, as that would be extremely dangerous. Accidentally modifying temporaries, only to have the temporaries evaporate along with your modifications, would lead to subtle and obnoxious bugs, so C++ rightly prohibits this. (I should mention that VC has an evil extension that allows this, but if you compile with /W4 , it warns when the evil extension is activated. Usually.) And it can't bind to const rvalues, as that would be doubly bad. (Careful readers should note that I'm not talking about template argument deduction here.)

const Type& binds to everything: modifiable lvalues, const lvalues, modifiable rvalues, and const rvalues (and can be used to observe them).

A reference is a name, so a reference bound to an rvalue is itself an lvalue (yes, L). (As only a const reference can be bound to an rvalue, it will be a const lvalue.) This is confusing, and will be an extremely big deal later, so I'll explain further. Given the function void observe(const string& str) , inside observe()'s implementation, str is a const lvalue, and its address can be taken and used before observe() returns. This is

true even though observe() can be called with rvalues, such as three() or four() above. observe("purr") can also be called, which constructs a temporary string and binds str to that temporary. The return values of three() and four() don't have names, so they're rvalues, but within observe(), str is a name, so it's an lvalue. As I said above, "lvalueness versus rvalueness is a property of expressions, not of objects". Of course, because str can be bound to a temporary which will evaporate, its address shouldn't be stored anywhere where it could be used after observe() returns.

Have you ever bound an rvalue to a const reference and then taken its address? Yes, you have! This is what happens when you write a copy assignment operator, Foo& operator=(const Foo& other) , with a self-assignment check, if (this != &other) { copy stuff; } return *this; , and you copy assign from a temporary, like Foo make_foo(); Foo f; f = make_foo(); .

At this point, you might ask, "So what's the difference between modifiable rvalues and const rvalues? I can't bind Type& to modifiable rvalues, and I can't assign things (etc.) to modifiable rvalues, so can I really modify them?" This is a very good question! In C++98/03, the answer is that there's a slight difference: non-const member functions can be called on modifiable rvalues. C++ doesn't want you to accidentally modify temporaries, but directly calling a non-const member function on a modifiable rvalue is explicit, so it's allowed. In C++0x, the answer changes dramatically, making move semantics possible.

Congratulations! Now you have what I call "lvalue/rvalue vision", the ability to look at an expression and determine whether it's an lvalue or an rvalue. Combined with your "const vision", you can precisely reason that given void mutate(string& ref) and the definitions above, mutate(one) is valid, while mutate(two), mutate(three()), mutate(four()), and mutate("purr") are invalid, and all of observe(one), observe(two), observe(three()), observe(four()), and observe("purr") are valid. If you're a C++98/03 programmer, you already knew which of these calls were valid and which were invalid; your "gut feeling", if not your compiler, would have told you that mutate(three()) was bogus. Your new lvalue/rvalue vision tells you precisely why (three() is an rvalue, and modifiable references can't be bound to rvalues). Is that useful? To language lawyers, yes, but not really to normal programmers. After all, you've gotten this far without knowing all of this stuff about lvalues and rvalues. But here's the catch: compared to C++98/03, C++0x has vastly more powerful lvalue/rvalue vision (in particular, the ability to look at an expression, determine

whether it's a modifiable/const lvalue/rvalue, and do something about it). In order to use C++0x effectively, you need lvalue/rvalue vision too. And now you have it, so we can proceed!

the copying problem

C++98/03 combines insanely powerful abstraction with insanely efficient execution, but it has a problem: it's overly fond of copying. Things with value semantics behave like ints, so copying a thing doesn't modify the source, and the resulting copies are independent. Value semantics are great, except that they tend to lead to unnecessary copies of heavy objects like strings, vectors, and so forth. ("Heavy" means "expensive to copy"; a million-element vector is heavy.) The Return Value Optimization (RVO) and Named Return Value Optimization (NRVO), where copy constructors are elided in certain situations, help to alleviate this problem, but they don't remove all unnecessary copies.

The *most unnecessary* copies are those where the source is about to be destroyed. Would you photocopy a sheet of paper and then immediately throw away the original, assuming that the original and the photocopy are identical? That would be wasteful; you should keep the original and not bother with the photocopy. Here's what I call "the killer example", derived from one of the Standardization Committee's examples (in N1377). Suppose that you have a bunch of strings, like this:

```
string s0("my mother told me that");

string s1("cute");

string s2("fluffy");
```

```
string s3("kittens");

string s4("are an essential part of a healthy diet");
```

And that you concatenate them like this:

```
string dest = s0 + " " + s1 + " " + s2 + " " + s3 + " " + s4;
```

How efficient is this? (We're not worrying about this specific example, which executes in microseconds; we're worrying about its generalization, which occurs throughout the entire language.)

Each call to `operator+()` returns a temporary `string`. There are 8 calls to `operator+()`, so there are 8 temporary `string`s. Each one, upon its construction, performs a dynamic memory allocation and copies all of the characters that have been concatenated so far, and later, upon its destruction, performs a dynamic memory deallocation. (If you've heard of the Small String Optimization, which VC performs in order to avoid dynamic memory allocations and deallocations for short `string`s, it's defeated here by my carefully chosen and sufficiently long `s0`, and even if it applied, it couldn't avoid the copying. If you've heard of the Copy-On-Write "optimization", forget about it – it doesn't apply here, and it's a pessimization under multithreading, so Standard Library implementations don't do it anymore.)

In fact, because every concatenation copies all of the characters that have been concatenated so far, this has quadratic complexity in the number of concatenations. Yuck! This is extraordinarily wasteful, which is especially embarrassing for C++. Why is this happening, and what can we do about it?

The problem is that `operator+()`, which takes two `const string&` or one `const string&`

and one const char * (there are other overloads, which we aren't using here), can't tell whether it's being fed lvalues versus rvalues, so it always has to create and return a new temporary string . Why do lvalues versus rvalues matter?

When evaluating s0 + " " , it's absolutely necessary to create a new temporary string . s0 is an lvalue, naming a persistent object, so we can't modify it. (Someone would notice!) But when evaluating (s0 + " ") + s1 , we could simply append s1's contents onto our first temporary string, instead of creating a second temporary and throwing the first temporary away. This is the **key insight behind move semantics**: because s0 + " " is an rvalue, an expression referring to a temporary object, no one else in the entire program can observe that temporary object. If we could detect that expression as being a modifiable rvalue, we could then proceed to modify the temporary object arbitrarily, without anyone else noticing. operator+() isn't "supposed to" modify its arguments, but if they're modifiable rvalues, who cares? In this manner, each call to operator+() can append characters onto a single temporary string . This completely eliminates the unnecessary dynamic memory management and unnecessary copying, leaving us with linear complexity. Yay!

Technically speaking, in C++0x, each call to operator+() still returns a separate temporary string . However, the second temporary string (from evaluating (s0 + " ") + s1) is constructed by stealing the memory owned by the first temporary string (from evaluating s0 + " ") and then appending s1's contents onto that memory (which may trigger an ordinary geometric reallocation). "Stealing" consists of pointer twiddling: the second temporary copies and then nulls out the first temporary's internal pointer. When the first temporary is eventually destroyed ("at the semicolon"), its pointer is null, so its destructor does nothing.

In general, being able to detect modifiable rvalues allows you to engage in "resource pilfering". If the objects referred to by modifiable rvalues own any resources (such as memory), you can steal their resources instead of copying them, since they're going to evaporate anyways. Constructing from or assigning from modifiable rvalues by taking what they own is generically referred to as "moving", and moveable objects have "move semantics".

This is extremely useful in many places, such as vector reallocation. When a vector needs more capacity (e.g. during push_back()) and undergoes reallocation, it needs to

copy elements from the old buffer into the new one. If the objects stored in the vector are expensive to copy (e.g. they could be vectors themselves), then this reallocation could be costly. This cost can be greatly reduced in many cases by *moving* the old elements into the new buffer rather than copying them. std::vector carries out this operation so long as the move construction of the elements is guaranteed to not throw an exception, as otherwise this could result in lost data.

Visual CPP Team

Follow

Posted in C++

Read next

MFC Restart Manager Support in VS2010

Hi, I am Weidong Huang, a Software Design Engineer in Test in the Visual C++ group. Today I am going to talk about MFC's support of Restart Manager ...

Visual CPP Team

February 18, 2009

0 comment

Quick Tips On Using Whole Program Optimization

Hi, I'm Jerry Goodwin from the Visual C++ code generation and optimization team, with a couple quick tips on using Whole Program Optimization, also referred to as ...

Visual CPP Team

February 24, 2009

0 comment

1 comment

Comments are closed. Login to edit/delete your existing comments

David Hanney
May 28, 2019 9:40 pm
0

Interesting and infomative article right up until it stops midsentence.
“This is extremely useful in many places, such as vector reallocation. When a vector needs more capacity (e.g. during push_back()) and undergoes reallocation, it needs to copy elements from” … from where? grrrrrrr

Relevant Links

Getting Started with C++ in VS
Bring Your Existing C++ Code to VS
C++ Code Editing & Navigation
C++ Unit Testing
C++ Debugging & Diagnostics
Collaborating with Your Team in VS
C++ Windows Development
C++ Linux Development
C++ Android & iOS Development
C++ Game Development

Topics

C++
Announcement
CMake
New Feature
Linux

Visual Studio Code
Diagnostics
General C++ Series
performance
Vcpkg
Writing Code

Archive

February 2023
January 2023
December 2022
November 2022
October 2022
September 2022
August 2022
July 2022
June 2022
May 2022
April 2022

Stay informed