What are move semantics?

I just finished listening to the Software Engineering radio podcast interview with Scott Meyers regarding C++0x. Most of the new features made sense to me, and I am actually excited about C++0x now, with the exception of one. I still don't get *move semantics*... What are they exactly?

c++ c++-faq c++11 move-semantics

edited Apr 2 '15 at 9:14
RJFalconer
4,949 ●3 ●27 ●50

asked Jun 23 '10 at 22:46
dicroce
19.7k ●22 ●82 ●126

10 I found [Eli Bendersky's blog article](eli.thegreenplace.net/2011/12/15/...) about lvalues and rvalues in C and C++ pretty informative. He also mentions rvalue references in C++11 and introduces them with small examples. – Nils Jul 18 '12 at 13:35

10 Alex Allain's exposition on the topic is very well written. – Patrick Sanan Dec 12 '13 at 19:47

11 Answers

I find it easiest to understand move semantics with example code. Let's start with a very simple string class which only holds a pointer to a heap-allocated block of memory:

```
#include <cstring>
#include <algorithm>

class string
{
    char* data;

public:

    string(const char* p)
    {
        size_t size = strlen(p) + 1;
        data = new char[size];
        memcpy(data, p, size);
    }
```

Since we chose to manage the memory ourselves, we need to follow the rule of three. I am going to defer writing the assignment operator and only implement the destructor and the copy constructor for now:

```
    ~string()
    {
        delete[] data;
    }

    string(const string& that)
    {
        size_t size = strlen(that.data) + 1;
        data = new char[size];
        memcpy(data, that.data, size);
    }
```

The copy constructor defines what it means to copy string objects. The parameter `const string& that` binds to all expressions of type string which allows you to make copies in the following examples:

```
string a(x);                                    // Line 1
string b(x + y);                                // Line 2
string c(some_function_returning_a_string());   // Line 3
```

Now comes the key insight into move semantics. Note that only in the first line where we copy `x` is this deep copy really necessary, because we might want to inspect `x` later and would be very surprised if `x` had changed somehow. Did you notice how I just said `x` three times (four times if you include this sentence) and meant the *exact same object* every time? We call expressions such as `x` "lvalues".

The arguments in lines 2 and 3 are not lvalues, but rvalues, because the underlying string objects have no names, so the client has no way to inspect them again at a later point in time. rvalues denote temporary objects which are destroyed at the next semicolon (to be more precise: at the end of the full-expression that lexically contains the rvalue). This is important because during the initialization of `b` and `c`, we could do whatever we wanted with the source string, and *the client couldn't tell a difference*!

C++0x introduces a new mechanism called "rvalue reference" which, among other things, allows us to detect rvalue arguments via function overloading. All we have to do is write a constructor with an rvalue reference parameter. Inside that constructor we can do *anything we want* with the source, as long as we leave it in *some* valid state:

```
    string(string&& that)   // string&& is an rvalue reference to a string
    {
        data = that.data;
        that.data = nullptr;
    }
```

What have we done here? Instead of deeply copying the heap data, we have just copied the pointer and then set the original pointer to null. In effect, we have "stolen" the data that originally belonged to the source string. Again, the key insight is that under no circumstance could the client detect that the source had been modified. Since we don't really do a copy here, we call this constructor a "move constructor". Its job is to move resources from one object to another instead of copying them.

Congratulations, you now understand the basics of move semantics! Let's continue by implementing the assignment operator. If you're unfamiliar with the copy and swap idiom, learn it and come back, because it's an awesome C++ idiom related to exception safety.

```
    string& operator=(string that)
    {
        std::swap(data, that.data);
        return *this;
    }
};
```

Huh, that's it? "Where's the rvalue reference?" you might ask. "We don't need it here!" is my answer :)

Note that we pass the parameter `that` *by value*, so `that` has to be initialized just like any other string object. Exactly how is `that` going to be initialized? In the olden days of C++98, the answer would have been "by the copy constructor". In C++0x, the compiler chooses between the copy constructor and the move constructor based on whether the argument to the assignment operator is an lvalue or an rvalue.

So if you say `a = b`, the *copy constructor* will initialize `that` (because the expression `b` is an lvalue), and the assignment operator swaps the contents with a freshly created, deep copy. That is the very definition of the copy and swap idiom -- make a copy, swap the contents with the copy, and then get rid of the copy by leaving the scope. Nothing new here.

But if you say `a = x + y`, the *move constructor* will initialize `that` (because the expression `x + y` is an rvalue), so there is no deep copy involved, only an efficient move. `that` is still an independent object from the argument, but its construction was trivial, since the heap data didn't have to be copied, just moved. It wasn't necessary to copy it because `x + y` is an rvalue, and again, it is okay to move from string objects denoted by rvalues.

To summarize, the copy constructor makes a deep copy, because the source must remain untouched. The move constructor, on the other hand, can just copy the pointer and then set the pointer in the source to null. It is okay to "nullify" the source object in this manner, because the client has no way of inspecting the object again.

I hope this example got the main point across. There is a lot more to rvalue references and move semantics which I intentionally left out to keep it simple. If you want more details please see my supplementary answer.

edited May 23 '17 at 12:10
Community ♦
1 ● 1

answered Jun 24 '10 at 12:40
fredoverflow
152k ● 60 ● 296 ● 578

64 Excellent answer, made it really clear. Any links to the things you left out? – Phil H Jun 15 '12 at 7:19

28 @But if my ctor is getting an rvalue, which can never be used later, why do I even need to bother leaving it in a consistent/safe state? Instead of setting that.data = 0, why not just leave it be? – einpoklum Jul 16 '13 at 14:03

50 @einpoklum Because without `that.data = 0`, the characters would be destroyed way too early (when the temporary dies), and also twice. You want to steal the data, not share it! – fredoverflow Jul 17 '13 at 17:36

10 @einpoklum The regularly-scheduled destructor still gets run, so you have to ensure that the source object's post-move state doesn't cause a crash. Better, you should make sure the source object can also be the receiver of an assignment or other write. – CTMacUser Sep 2 '13 at 10:49

6 @pranitkothari Yes, all objects must be destructed, even moved-from objects. And since we do not want the char array to be deleted when that happens, we have to set the pointer to null. – fredoverflow Oct 3 '13 at 7:55

My first answer was an extremely simplified introduction to move semantics, and many details were left out on purpose to keep it simple. However, there is a lot more to move semantics, and I thought it was time for a second answer to fill the gaps. The first answer is already quite old, and it did not feel right to simply replace it with a completely different text. I think it still serves well as a first introduction. But if you want to dig deeper, read on :)

Stephan T. Lavavej took the time provide valuable feedback. Thank you very much, Stephan!

Introduction

Move semantics allows an object, under certain conditions, to take ownership of some other object's external resources. This is important in two ways:

1. Turning expensive copies into cheap moves. See my first answer for an example. Note that if an object does not manage at least one external resource (either directly, or indirectly through its member objects), move semantics will not offer any advantages over copy semantics. In that case, copying an object and moving an object means the exact same thing:

   ```
   class cannot_benefit_from_move_semantics
   {
       int a;        // moving an int means copying an int
       float b;      // moving a float means copying a float
       double c;     // moving a double means copying a double
       char d[64];   // moving a char array means copying a char array

       // ...
   };
   ```

2. Implementing safe "move-only" types; that is, types for which copying does not make sense, but moving does. Examples include locks, file handles, and smart pointers with unique ownership semantics. Note: This answer discusses `std::auto_ptr`, a deprecated C++98 standard library template, which was replaced by `std::unique_ptr` in C++11. Intermediate C++ programmers are probably at least somewhat familiar with `std::auto_ptr`, and because of the "move semantics" it displays, it seems like a good starting point for discussing move semantics in C++11. YMMV.

What is a move?

The C++98 standard library offers a smart pointer with unique ownership semantics called `std::auto_ptr<T>`. In case you are unfamiliar with `auto_ptr`, its purpose is to guarantee that a dynamically allocated object is always released, even in the face of exceptions:

```
{
    std::auto_ptr<Shape> a(new Triangle);
    // ...
    // arbitrary code, could throw exceptions
    // ...
}   // <--- when a goes out of scope, the triangle is deleted automatically
```

The unusual thing about `auto_ptr` is its "copying" behavior:

```
auto_ptr<Shape> a(new Triangle);

      +---------------+
      | triangle data |
      +---------------+
        ^
        |
        |
        |
  +-----|---+
  |   +-|-+ |
a | p | | | |
  |   +---+ |
  +---------+

auto_ptr<Shape> b(a);

      +---------------+
      | triangle data |
      +---------------+
        ^
        |
        +----------------------+
                               |
  +---------+            +-----|---+
  |   +---+ |            |   +-|-+ |
a | p |   | |          b | p | | | |
  |   +---+ |            |   +---+ |
  +---------+            +---------+
```

Note how the initialization of `b` with `a` does *not* copy the triangle, but instead transfers the ownership of the triangle from `a` to `b`. We also say "`a` is *moved into* `b`" or "the triangle is *moved* from `a` *to* `b`". This may sound confusing, because the triangle itself always stays at the same place in memory.

> To move an object means to transfer ownership of some resource it manages to another object.

The copy constructor of `auto_ptr` probably looks something like this (somewhat simplified):

```
auto_ptr(auto_ptr& source)   // note the missing const
{
    p = source.p;
    source.p = 0;   // now the source no longer owns the object
}
```

Dangerous and harmless moves

The dangerous thing about `auto_ptr` is that what syntactically looks like a copy is actually a move. Trying to call a member function on a moved-from `auto_ptr` will invoke undefined behavior, so you have to be very careful not to use an `auto_ptr` after it has been moved from:

```
auto_ptr<Shape> a(new Triangle);   // create triangle
auto_ptr<Shape> b(a);              // move a into b
double area = a->area();           // undefined behavior
```

But `auto_ptr` is not *always* dangerous. Factory functions are a perfectly fine use case for `auto_ptr`:

```
auto_ptr<Shape> make_triangle()
{
    return auto_ptr<Shape>(new Triangle);
}

auto_ptr<Shape> c(make_triangle());      // move temporary into c
double area = make_triangle()->area();   // perfectly safe
```

Note how both examples follow the same syntactic pattern:

```
auto_ptr<Shape> variable(expression);
double area = expression->area();
```

And yet, one of them invokes undefined behavior, whereas the other one does not. So what is the difference between the expressions `a` and `make_triangle()`? Aren't they both of the same type? Indeed they are, but they have different *value categories*.

Value categories

Obviously, there must be some profound difference between the expression `a` which denotes an `auto_ptr` variable, and the expression `make_triangle()` which denotes the call of a function that returns an `auto_ptr` by value, thus creating a fresh temporary `auto_ptr` object every time it is called. `a` is an example of an *lvalue*, whereas `make_triangle()` is an example of an *rvalue*.

Moving from lvalues such as `a` is dangerous, because we could later try to call a member function via `a`, invoking undefined behavior. On the other hand, moving from rvalues such as `make_triangle()` is perfectly safe, because after the copy constructor has done its job, we cannot use the temporary again. There is no expression that denotes said temporary; if we simply write `make_triangle()` again, we get a *different* temporary. In fact, the moved-from temporary is already gone on the next line:

```
auto_ptr<Shape> c(make_triangle());
                                  ^ the moved-from temporary dies right here
```

Note that the letters `l` and `r` have a historic origin in the left-hand side and right-hand side of an assignment. This is no longer true in C++, because there are lvalues which cannot appear on the left-hand side of an assignment (like arrays or user-defined types without an assignment operator), and there are rvalues which can (all rvalues of class types with an assignment operator).

> An rvalue of class type is an expression whose evaluation creates a temporary object. Under normal circumstances, no other expression inside the same scope denotes the same temporary object.

Rvalue references

We now understand that moving from lvalues is potentially dangerous, but moving from rvalues is harmless. If C++ had language support to distinguish lvalue arguments from rvalue arguments, we could either completely forbid moving from lvalues, or at least make moving from lvalues *explicit* at call site, so that we no longer move by accident.

C++11's answer to this problem is *rvalue references*. An rvalue reference is a new kind of reference that only binds to rvalues, and the syntax is `X&&`. The good old reference `X&` is now known as an *lvalue reference*. (Note that `X&&` is *not* a reference to a reference; there is no such thing in C++.)

If we throw `const` into the mix, we already have four different kinds of references. What kinds of expressions of type `X` can they bind to?

```
            lvalue   const lvalue   rvalue   const rvalue
---------------------------------------------------------
X&          yes
const X&    yes      yes            yes      yes
X&&                                 yes
const X&&                           yes      yes
```

In practice, you can forget about `const X&&`. Being restricted to read from rvalues is not very useful.

> An rvalue reference `X&&` is a new kind of reference that only binds to rvalues.

Implicit conversions

Rvalue references went through several versions. Since version 2.1, an rvalue reference `X&&` also binds to all value categories of a different type `Y`, provided there is an implicit conversion from `Y` to `X`. In that case, a temporary of type `X` is created, and the rvalue reference is bound to that temporary:

```
void some_function(std::string&& r);

some_function("hello world");
```

In the above example, `"hello world"` is an rvalue of type `const char[12]`. Since there is an implicit conversion from `const char[12]` through `const char*` to `std::string`, a temporary of type `std::string` is created, and `r` is bound to that temporary. This is one of the cases where the distinction between rvalues (expressions) and temporaries (objects) is a bit blurry.

Move constructors

A useful example of a function with an `X&&` parameter is the *move constructor* `X::X(X&& source)`. Its purpose is to transfer ownership of the managed resource from the source into the current object.

In C++11, `std::auto_ptr<T>` has been replaced by `std::unique_ptr<T>` which takes advantage of rvalue references. I will develop and discuss a simplified version of `unique_ptr`. First, we encapsulate a raw pointer and overload the operators `->` and `*`, so our class feels like a pointer:

```
template<typename T>
class unique_ptr
{
    T* ptr;

public:

    T* operator->() const
    {
        return ptr;
    }

    T& operator*() const
    {
        return *ptr;
    }
```

The constructor takes ownership of the object, and the destructor deletes it:

```
    explicit unique_ptr(T* p = nullptr)
    {
        ptr = p;
    }

    ~unique_ptr()
    {
        delete ptr;
    }
```

Now comes the interesting part, the move constructor:

```
    unique_ptr(unique_ptr&& source)   // note the rvalue reference
    {
        ptr = source.ptr;
        source.ptr = nullptr;
    }
```

This move constructor does exactly what the `auto_ptr` copy constructor did, but it can only be supplied with rvalues:

```
unique_ptr<Shape> a(new Triangle);
unique_ptr<Shape> b(a);                 // error
unique_ptr<Shape> c(make_triangle());   // okay
```

The second line fails to compile, because `a` is an lvalue, but the parameter `unique_ptr&& source` can only be bound to rvalues. This is exactly what we wanted; dangerous moves should never be implicit. The third line compiles just fine, because `make_triangle()` is an rvalue. The move constructor will transfer ownership from the temporary to `c`. Again, this is exactly what we wanted.

> The move constructor transfers ownership of a managed resource into the current object.

Move assignment operators

The last missing piece is the move assignment operator. Its job is to release the old resource and acquire the new resource from its argument:

```
    unique_ptr& operator=(unique_ptr&& source)   // note the rvalue reference
    {
        if (this != &source)    // beware of self-assignment
        {
            delete ptr;         // release the old resource

            ptr = source.ptr;   // acquire the new resource
            source.ptr = nullptr;
        }
        return *this;
    }
};
```

Note how this implementation of the move assignment operator duplicates logic of both the destructor and the move constructor. Are you familiar with the copy-and-swap idiom? It can also be applied to move semantics as the move-and-swap idiom:

```
    unique_ptr& operator=(unique_ptr source)   // note the missing reference
    {
        std::swap(ptr, source.ptr);
        return *this;
    }
};
```

Now that `source` is a variable of type `unique_ptr`, it will be initialized by the move constructor; that is, the argument will be moved into the parameter. The argument is still required to be an rvalue, because the move constructor itself has an rvalue reference parameter. When control flow reaches the closing brace of `operator=`, `source` goes out of scope, releasing the old resource automatically.

The move assignment operator transfers ownership of a managed resource into the current object, releasing the old resource. The move-and-swap idiom simplifies the implementation.

Moving from lvalues

Sometimes, we want to move from lvalues. That is, sometimes we want the compiler to treat an lvalue as if it were an rvalue, so it can invoke the move constructor, even though it could be potentially unsafe. For this purpose, C++11 offers a standard library function template called `std::move` inside the header `<utility>`. This name is a bit unfortunate, because `std::move` simply casts an lvalue to an rvalue; it does *not* move anything by itself. It merely *enables* moving. Maybe it should have been named `std::cast_to_rvalue` or `std::enable_move`, but we are stuck with the name by now.

Here is how you explicitly move from an lvalue:

```
unique_ptr<Shape> a(new Triangle);
unique_ptr<Shape> b(a);              // still an error
unique_ptr<Shape> c(std::move(a));   // okay
```

Note that after the third line, `a` no longer owns a triangle. That's okay, because by *explicitly* writing `std::move(a)`, we made our intentions clear: "Dear constructor, do whatever you want with `a` in order to initialize `c`; I don't care about `a` anymore. Feel free to have your way with `a`."

`std::move(some_lvalue)` casts an lvalue to an rvalue, thus enabling a subsequent move.

Xvalues

Note that even though `std::move(a)` is an rvalue, its evaluation does *not* create a temporary object. This conundrum forced the committee to introduce a third value category. Something that can be bound to an rvalue reference, even though it is not an rvalue in the traditional sense, is called an *xvalue* (eXpiring value). The traditional rvalues were renamed to *prvalues* (Pure rvalues).

Both prvalues and xvalues are rvalues. Xvalues and lvalues are both *glvalues* (Generalized lvalues). The relationships are easier to grasp with a diagram:

```
        expressions
          /     \
         /       \
        /         \
    glvalues   rvalues
      /  \       /  \
     /    \     /    \
    /      \   /      \
lvalues   xvalues   prvalues
```

Note that only xvalues are really new; the rest is just due to renaming and grouping.

C++98 rvalues are known as prvalues in C++11. Mentally replace all occurrences of "rvalue" in the preceding paragraphs with "prvalue".

Moving out of functions

So far, we have seen movement into local variables, and into function parameters. But moving is also possible in the opposite direction. If a function returns by value, some object at call site (probably a local variable or a temporary, but could be any kind of object) is initialized with the expression after the `return` statement as an argument to the move constructor:

```
unique_ptr<Shape> make_triangle()
{
    return unique_ptr<Shape>(new Triangle);
}          \----------------------------/
                  |
                  | temporary is moved into c
                  |
                  v
unique_ptr<Shape> c(make_triangle());
```

Perhaps surprisingly, automatic objects (local variables that are not declared as `static`) can also be *implicitly* moved out of functions:

```
unique_ptr<Shape> make_square()
{
    unique_ptr<Shape> result(new Square);
    return result;   // note the missing std::move
}
```

How come the move constructor accepts the lvalue `result` as an argument? The scope of `result` is about to end, and it will be destroyed during stack unwinding. Nobody could possibly complain afterwards that `result` had changed somehow; when control flow is back at the caller, `result` does not exist anymore! For that reason, C++11 has a special rule that allows returning automatic objects from functions without having to write `std::move`. In fact, you should *never* use `std::move` to move automatic objects out of functions, as this inhibits the "named return value optimization" (NRVO).

> Never use `std::move` to move automatic objects out of functions.

Note that in both factory functions, the return type is a value, not an rvalue reference. Rvalue references are still references, and as always, you should never return a reference to an automatic object; the caller would end up with a dangling reference if you tricked the compiler into accepting your code, like this:

```
unique_ptr<Shape>&& flawed_attempt()   // DO NOT DO THIS!
{
    unique_ptr<Shape> very_bad_idea(new Square);
    return std::move(very_bad_idea);   // WRONG!
}
```

> Never return automatic objects by rvalue reference. Moving is exclusively performed by the move constructor, not by `std::move`, and not by merely binding an rvalue to an rvalue reference.

Moving into members

Sooner or later, you are going to write code like this:

```
class Foo
{
    unique_ptr<Shape> member;

public:

    Foo(unique_ptr<Shape>&& parameter)
    : member(parameter)   // error
    {}
};
```

Basically, the compiler will complain that `parameter` is an lvalue. If you look at its type, you see an rvalue reference, but an rvalue reference simply means "a reference that is bound to an rvalue"; it does *not* mean that the reference itself is an rvalue! Indeed, `parameter` is just an ordinary variable with a name. You can use `parameter` as often as you like inside the body of the constructor, and it always denotes the same object. Implicitly moving from it would be dangerous, hence the language forbids it.

> A named rvalue reference is an lvalue, just like any other variable.

The solution is to manually enable the move:

```
class Foo
{
    unique_ptr<Shape> member;

public:

    Foo(unique_ptr<Shape>&& parameter)
    : member(std::move(parameter))   // note the std::move
    {}
};
```

You could argue that `parameter` is not used anymore after the initialization of `member`. Why is there no special rule to silently insert `std::move` just as with return values? Probably because it would be too much burden on the compiler implementors. For example, what if the constructor body was in another translation unit? By contrast, the return value rule simply has to check the symbol tables to determine whether or not the identifier after the `return` keyword denotes an automatic object.

You can also pass `parameter` by value. For move-only types like `unique_ptr`, it seems there is no established idiom yet. Personally, I prefer pass by value, as it causes less clutter in the interface.

Special member functions

C++98 implicitly declares three special member functions on demand, that is, when they are needed somewhere: the copy constructor, the copy assignment operator and the destructor.

```
X::X(const X&);              // copy constructor
X& X::operator=(const X&);   // copy assignment operator
X::~X();                     // destructor
```

Rvalue references went through several versions. Since version 3.0, C++11 declares two additional special member functions on demand: the move constructor and the move assignment operator. Note that neither VC10 nor VC11 conform to version 3.0 yet, so you will have to implement them yourself.

```
X::X(X&&);                   // move constructor
X& X::operator=(X&&);        // move assignment operator
```

These two new special member functions are only implicitly declared if none of the special member functions are declared manually. Also, if you declare your own move constructor or move assignment operator, neither the copy constructor nor the copy assignment operator will be declared implicitly.

What do these rules mean in practice?

> If you write a class without unmanaged resources, there is no need to declare any of the five special member functions yourself, and you will get correct copy semantics and move semantics for free. Otherwise, you will have to implement the special member functions yourself. Of course, if your class does not benefit from move semantics, there is no need to implement the special move operations.

Note that the copy assignment operator and the move assignment operator can be fused into a single, unified assignment operator, taking its argument by value:

```
X& X::operator=(X source)    // unified assignment operator
{
    swap(source);            // see my first answer for an explanation
    return *this;
}
```

This way, the number of special member functions to implement drops from five to four. There is a tradeoff between exception-safety and efficiency here, but I am not an expert on this issue.

Forwarding references (previously known as *Universal references*)

Consider the following function template:

```
template<typename T>
void foo(T&&);
```

You might expect `T&&` to only bind to rvalues, because at first glance, it looks like an rvalue reference. As it turns out though, `T&&` also binds to lvalues:

```
foo(make_triangle());   // T is unique_ptr<Shape>, T&& is unique_ptr<Shape>&&
unique_ptr<Shape> a(new Triangle);
foo(a);                 // T is unique_ptr<Shape>&, T&& is unique_ptr<Shape>&
```

If the argument is an rvalue of type `X`, `T` is deduced to be `X`, hence `T&&` means `X&&`. This is what anyone would expect. But if the argument is an lvalue of type `X`, due to a special rule, `T` is deduced to be `X&`, hence `T&&` would mean something like `X& &&`. But since C++ still has no notion of references to references, the type `X& &&` is *collapsed* into `X&`. This may sound confusing and useless at first, but reference collapsing is essential for *perfect forwarding* (which will not be discussed here).

> T&& is not an rvalue reference, but a forwarding reference. It also binds to lvalues, in which case `T` and `T&&` are both lvalue references.

If you want to constrain a function template to rvalues, you can combine SFINAE with type traits:

```
#include <type_traits>

template<typename T>
typename std::enable_if<std::is_rvalue_reference<T&&>::value, void>::type
foo(T&&);
```

Implementation of move

Now that you understand reference collapsing, here is how `std::move` is implemented:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& t)
{
    return static_cast<typename std::remove_reference<T>::type&&>(t);
}
```

As you can see, `move` accepts any kind of parameter thanks to the forwarding reference `T&&`, and it returns an rvalue reference. The `std::remove_reference<T>::type` meta-function call is necessary because otherwise, for lvalues of type `X`, the return type would be `X& &&`, which would collapse into `X&`. Since `t` is always an lvalue (remember that a named rvalue reference is an lvalue), but we want to bind `t` to an rvalue reference, we have to explicitly cast `t` to the correct return type. The call of a function that returns an rvalue reference is itself an xvalue. Now you know where xvalues come from ;)

> The call of a function that returns an rvalue reference, such as `std::move`, is an xvalue.

Note that returning by rvalue reference is fine in this example, because `t` does not denote an automatic object, but instead an object that was passed in by the caller.

edited Jun 26 '17 at 1:34
ziad-saab
8,780 ● 1 ● 22 ● 24

answered Jul 18 '12 at 11:24
fredoverflow
152k ● 60 ● 296 ● 578

149 "It's only 10 pages on my monitor" – Mooing Duck Jul 18 '12 at 18:02

20 There's a third reason move semantics are important: exception safety. Often where a copy operation may throw (because it needs to allocate resources and the allocation may fail) a move operation can be no-throw (because it can transfer ownership of existing resources instead of allocating new ones). Having operations that can't fail is always nice, and it can be crucial when writing code that provides exception guarantees. – Brangdon Aug 5 '12 at 14:28

5 I was with you right up to 'Universal references', but then it's all too abstract to follow. Reference collapsing? Perfect forwarding? Are you saying that an rvalue reference becomes a universal reference if the type is templated? I wish there was a way to explain this so that I would know if I need to understand it or not! :) – Kylotan Nov 7 '14 at 10:56

3 Please write a book now ... this answer has given me reason to believe if you covered other corners of C++ in a lucid manner like this, thousands of more people will understand it. – halivingston Sep 27 '15 at 23:11

9 @halivingston Thank you very much for your kind feedback, I really appreciate it. The problem with writing a book is: it's much more work than you can possibly imagine. If you want to dig deep into C++11 and beyond, I suggest you buy "Effective Modern C++" by Scott Meyers. – fredoverflow Sep 28 '15 at 8:45

Move semantics are based on ***rvalue references***.
An rvalue is a temporary object, which is going to be destroyed at the end of the expression. In current C++, rvalues only bind to `const` references. C++1x will allow non-`const` rvalue references, spelled `T&&`, which are references to an rvalue objects.
Since an rvalue is going to die at the end of an expression, you can *steal its data*. Instead of *copying* it into another object, you *move* its data into it.

```
class X {
public:
  X(X&& rhs) // ctor taking an rvalue reference, so-called move-ctor
    : data_()
  {
     // since 'x' is an rvalue object, we can steal its data
     this->swap(std::move(rhs));
     // this will leave rhs with the empty data
  }
  void swap(X&& rhs);
  // ...
};

// ...

X f();

X x = f(); // f() returns result as rvalue, so this calls move-ctor
```

In the above code, with old compilers the result of `f()` is ***copied*** into `x` using `X`'s copy constructor. If your compiler supports move semantics and `X` has a move-constructor, then that is called instead.

Since its `rhs` argument is an *rvalue*, we know it's not needed any longer and we can steal its value. So the value is ***moved*** from the unnamed temporary returned from `f()` to `x` (while the data of `x`, initialized to an empty `X`, is moved into the temporary, which will get destroyed after the assignment).

edited Nov 24 '17 at 15:23

answered Jun 23 '10 at 23:12

sbi
155k ● 37 ● 204 ● 386

1 note that it should be `this->swap(std::move(rhs));` because named rvalue references are lvalues – wmamrak Jan 14 '14 at 15:22

This is kinda wrong, per @Tacyt's comment: `rhs` is an **lvalue** in the context of `X::X(X&& rhs)`. You need to call `std::move(rhs)` to get an rvalue, but this kinda makes the answer moot. – Ashe Apr 5 '14 at 3:05

Suppose you have a function that returns a substantial object:

```
Matrix multiply(const Matrix &a, const Matrix &b);
```

When you write code like this:

```
Matrix r = multiply(a, b);
```

then an ordinary C++ compiler will create a temporary object for the result of `multiply()`, call the copy constructor to initialise `r`, and then destruct the temporary return value. Move semantics in C++0x allow the "move constructor" to be called to initialise `r` by copying its contents, and then discard the temporary value without having to destruct it.

This is especially important if (like perhaps the `Matrix` example above), the object being copied allocates extra memory on the heap to store its internal representation. A copy constructor would have to either make a full copy of the internal representation, or use reference counting and copy-on-write semantics interally. A move constructor would leave the heap memory alone and just copy the pointer inside the `Matrix` object.

answered Jun 23 '10 at 22:53

Greg Hewgill
609k ● 130 ● 977 ● 1127

1 How are move constructors and copy constructors different? – dicroce Jun 23 '10 at 22:55

1 @dicroce: They differ by syntax, one looks like Matrix(const Matrix& src) (copy constructor) and the other looks like Matrix(Matrix&& src) (move constructor), check my main answer for a better example. – snk_kid Jun 23 '10 at 23:00 ✎

3 @dicroce: One makes a blank object, and one makes a copy. If the data stored in the object is large, a copy can be expensive. For example, std::vector. – Billy ONeal Jun 24 '10 at 2:49

1 @kunj2aan: It depends on your compiler, I suspect. The compiler could create a temporary object inside the function, and then move it into the caller's return value. Or, it may be able to directly construct the object in the return value, without needing to use a move constructor. – Greg Hewgill Jun 14 '12 at 19:44

1 @Jichao: That is an optimisation called RVO, see this question for more information on the difference: stackoverflow.com/questions/5031778/… – Greg Hewgill Oct 10 '13 at 8:20

If you are really interested in a good, in-depth explanation of move semantics, I'd highly recommend reading the original paper on them, "A Proposal to Add Move Semantics Support to the C++ Language."

It's very accessible and easy to read and it makes an excellent case for the benefits that they offer. There are other more recent and up to date papers about move semantics available on the WG21 website, but this one is probably the most straightforward since it approaches things from a top-level view and doesn't get very much into the gritty language details.

answered Jun 23 '10 at 23:32

James McNellis
273k ● 53 ● 775 ● 886

Move semantics is about **transferring resources rather than copying them** when nobody needs the source value anymore.

In C++03, objects are often copied, only to be destroyed or assigned-over before any code uses the value again. For example, when you return by value from a function—unless RVO kicks in—the value you're returning is copied to the caller's stack frame, and then it goes out of scope and is destroyed. This is just one of many examples: see pass-by-value when the source object is a temporary, algorithms like `sort` that just rearrange items, reallocation in `vector` when its `capacity()` is exceeded, etc.

When such copy/destroy pairs are expensive, it's typically because the object owns some heavyweight resource. For example, `vector<string>` may own a dynamically-allocated memory block containing an array of `string` objects, each with its own dynamic memory. Copying such an object is costly: you have to allocate new memory for each dynamically-allocated blocks in the source, and copy all the values across. *Then* you need deallocate all that memory you just copied. However, *moving* a large `vector<string>` means just copying a few pointers (that refer to the dynamic memory block) to the destination and zeroing them out in the source.

answered Apr 8 '12 at 19:47

Dave Abrahams
5,904 ● 5 ● 24 ● 18

In easy (practical) terms:

Copying an object means copying its "static" members and calling the `new` operator for its dynamic objects. Right?

```
class A
{
   int i, *p;

public:
   A(const A& a) : i(a.i), p(new int(*a.p)) {}
   ~A() { delete p; }
};
```

However, to **move** an object (I repeat, in a practical point of view) implies only to copy the pointers of dynamic objects, and not to create new ones.

But, is that not dangerous? Of course, you could destruct a dynamic object twice (segmentation fault). So, to avoid that, you should "invalidate" the source pointers to avoid destructing them twice:

```
class A
{
   int i, *p;

public:
   // Movement of an object inside a copy constructor.
   A(const A& a) : i(a.i), p(a.p)
   {
     a.p = nullptr; // pointer invalidated.
   }

   ~A() { delete p; }
   // Deleting NULL, 0 or nullptr (address 0x0) is safe.
};
```

Ok, but if I move an object, the source object becomes useless, no? Of course, but in certain situations that's very useful. The most evident one is when I call a function with an anonymous object (temporal, rvalue object, ..., you can call it with different names):

```
void heavyFunction(HeavyType());
```

In that situation, an anonymous object is created, next copied to the function parameter, and afterwards deleted. So, here it is better to move the object, because you don't need the anonymous object and you can save time and memory.

This leads to the concept of an "rvalue" reference. They exist in C++11 only to detect if the received object is anonymous or not. I think you do already know that an "lvalue" is an assignable entity (the left part of the `=` operator), so you need a named reference to an object to be capable to act as an lvalue. A rvalue is exactly the opposite, an object with no named references. Because of that, anonymous object and rvalue are synonyms. So:

```
class A
{
   int i, *p;

public:
   // Copy
   A(const A& a) : i(a.i), p(new int(*a.p)) {}

   // Movement (&& means "rvalue reference to")
   A(A&& a) : i(a.i), p(a.p)
   {
      a.p = nullptr;
   }

   ~A() { delete p; }
};
```

In this case, when an object of type `A` should be "copied", the compiler creates a lvalue reference or a rvalue reference according to if the passed object is named or not. When not, your move-constructor is called and you know the object is temporal and you can move its dynamic objects instead of copying them, saving space and memory.

It is important to remember that "static" objects are always copied. There's no ways to "move" a static object (object in stack and not on heap). So, the distinction "move"/ "copy" when an object has no dynamic members (directly or indirectly) is irrelevant.

If your object is complex and the destructor has other secondary effects, like calling to a library's function, calling to other global functions or whatever it is, perhaps is better to signal a movement with a flag:

```
class Heavy
{
   bool b_moved;
   // staff

public:
   A(const A& a) { /* definition */ }
   A(A&& a) : // initialization list
   {
      a.b_moved = true;
   }

   ~A() { if (!b_moved) /* destruct object */ }
};
```

So, your code is shorter (you don't need to do a `nullptr` assignment for each dynamic member) and more general.

Other typical question: what is the difference between `A&&` and `const A&&`? Of course, in the first case, you can modify the object and in the second not, but, practical meaning? In the second case, you can't modify it, so you have no ways to invalidate the object (except with a mutable flag or something like that), and there is no practical difference to a copy constructor.

And what is **perfect forwarding**? It is important to know that a "rvalue reference" is a reference to a named object in the "caller's scope". But in the actual scope, a rvalue reference is a name to an object, so, it acts as a named object. If you pass an rvalue reference to another function, you are passing a named object, so, the object isn't received like a temporal object.

```
void some_function(A&& a)
{
   other_function(a);
}
```

The object `a` would be copied to the actual parameter of `other_function`. If you want the object `a` continues being treated as a temporary object, you should use the `std::move` function:

```
other_function(std::move(a));
```

With this line, `std::move` will cast `a` to an rvalue and `other_function` will receive the object as a unnamed object. Of course, if `other_function` has not specific overloading to work with unnamed objects, this distinction is not important.

Is that perfect forwarding? Not, but we are very close. Perfect forwarding is only useful to work with templates, with the purpose to say: if I need to pass an object to another function, I need that if I receive a named object, the object is passed as a named object, and when not, I want to pass it like a unnamed object:

```
template<typename T>
void some_function(T&& a)
{
   other_function(std::forward<T>(a));
}
```

That's the signature of a prototypical function that uses perfect forwarding, implemented in C++11 by means of `std::forward`. This function exploits some rules of template instantiation:

```
 `A& && == A&`
 `A&& && == A&&`
```

So, if `T` is a lvalue reference to `A` (**T** = A&), `a` also (**A&** && => A&). If `T` is a rvalue reference to `A`, `a` also (A&& && => A&&). In both cases, `a` is a named object in the actual scope, but `T` contains the information of its "reference type" from the caller scope's point of view. This information (`T`) is passed as template parameter to `forward` and 'a' is moved or not according to the type of `T`.

edited Feb 20 '15 at 23:09
simbo1905
2,761 ● 24 ● 43

answered Aug 18 '13 at 15:57
Peregring-lk
3,834 ● 2 ● 22 ● 54

It's like copy semantics, but instead of having to duplicate all of the data you get to steal the data from the object being "moved" from.

edited Sep 27 '13 at 7:48
gd1
8,065 ●6 ●30 ●73

answered Jun 23 '10 at 22:56
Terry Mahaffey
9,677 ●26 ●41

You know what a copy semantics means right? it means you have types which are copyable, for user-defined types you define this either buy explicitly writing a copy constructor & assignment operator or the compiler generates them implicitly. This will do a copy.

Move semantics is basically a user-defined type with constructor that takes an r-value reference (new type of reference using && (yes two ampersands)) which is non-const, this is called a move constructor, same goes for assignment operator. So what does a move constructor do, well instead of copying memory from it's source argument it 'moves' memory from the source to the destination.

When would you want to do that? well std::vector is an example, say you created a temporary std::vector and you return it from a function say:

```
std::vector<foo> get_foos();
```

You're going to have overhead from the copy constructor when the function returns, if (and it will in C++0x) std::vector has a move constructor instead of copying it can just set it's pointers and 'move' dynamically allocated memory to the new instance. It's kind of like transfer-of-ownership semantics with std::auto_ptr.

edited Jun 23 '10 at 23:03

answered Jun 23 '10 at 22:58
snk_kid
2,752 ●1 ●18 ●18

1 I don't think this is a great example, because in these function return value examples the Return Value Optimization is probably already eliminating the copy operation. – Zan Lynx Jun 28 '10 at 18:32

To illustrate the need for *move semantics*, let's consider this example without move semantics:

Here's a function that takes an object of type `T` and returns an object of the same type `T`:

```
T f(T o) { return o; }
  //^^^ new object constructed
```

The above function uses *call by value* which means that when this function is called an object must be *constructed* to be used by the function.
Because the function also *returns by value*, another new object is constructed for the return value:

```
T b = f(a);
  //^ new object constructed
```

Two new objects have been constructed, one of which is a temporary object that's only used for the duration of the function.

When the new object is created from the return value, the copy constructor is called to *copy* the contents of the temporary object to the new object b. After the function completes, the temporary object used in the function goes out of scope and is destroyed.

Now, let's consider what a *copy constructor* does.

It must first initialize the object, then copy all the relevant data from the old object to the new one. Depending on the class, maybe its a container with very much data, then that could represent much *time* and *memory usage*

```
// Copy constructor
T::T(T &old) {
    copy_data(m_a, old.m_a);
    copy_data(m_b, old.m_b);
    copy_data(m_c, old.m_c);
}
```

With **move semantics** it's now possible to make most of this work less unpleasant by simply *moving* the data rather than copying.

```
// Move constructor
T::T(T &&old) noexcept {
    m_a = std::move(old.m_a);
    m_b = std::move(old.m_b);
    m_c = std::move(old.m_c);
}
```

Moving the data involves re-associating the data with the new object. And *no copy takes place* at all.

This is accomplished with an `rvalue` reference.
An `rvalue` reference works pretty much like an `lvalue` reference with one important difference:
an *rvalue reference can be moved* and an *lvalue* cannot.

From cppreference.com:

> To make strong exception guarantee possible, user-defined move constructors should not throw exceptions. In fact, standard containers typically rely on std::move_if_noexcept to choose between move and copy when container elements need to be relocated. If both copy and move constructors are provided, overload resolution selects the move constructor if the argument is an rvalue (either a prvalue such as a nameless temporary or an xvalue such as the result of std::move), and selects the copy constructor if the argument is an lvalue (named object or a function/operator returning lvalue reference). If only the copy constructor is provided, all argument categories select it (as long as it takes a reference to const, since rvalues can bind to const references), which makes copying the fallback for moving, when moving is unavailable. In many situations, move constructors are optimized out even if they would produce observable side-effects, see copy elision. A constructor is called a 'move constructor' when it takes an rvalue reference as a parameter. It is not obligated to move anything, the class is not required to have a resource to be moved and a 'move constructor' may not be able to move a resource as in the allowable (but maybe not sensible) case where the parameter is a const rvalue reference (const T&&).

edited Feb 25 '16 at 0:12

answered Feb 25 '16 at 0:00
Andreas DM
5,673 ● 5 ● 20 ● 44

I'm writing this to make sure I understand it properly.

Move semantics were created to avoid the unnecessary copying of large objects. Bjarne Stroustrup in his book "The C++ Programming Language" uses two examples where unnecessary copying occurs by default: one, the swapping of two large objects, and two, the returning of a large object from a method.

Swapping two large objects usually involves copying the first object to a temporary object, copying the second object to the first object, and copying the temporary object to the second object. For a built-in type, this is very fast, but for large objects these three copies could take a large amount of time. A "move assignment" allows the programmer to override the default copy behavior and instead swap references to the objects, which means that there is no copying at all and the swap operation is much faster. The move assignment can be invoked by calling the std::move() method.

Returning an object from a method by default involves making a copy of the local object and its associated data in a location which is accessible to the caller (because the local object is not accessible to the caller and disappears when the method finishes). When a built-in type is being returned, this operation is very fast, but if a large object is being returned, this could take a long time. The move constructor allows the programmer to override this default behavior and instead "reuse" the heap data associated with the local object by pointing the object being returned to the caller to heap data associated with the local object. Thus no copying is required.

In languages which do not allow the creation of local objects (that is, objects on the stack) these types of problems do not occur as all objects are allocated on the heap and are always accessed by reference.

edited Nov 22 '16 at 5:23

answered Nov 18 '16 at 23:12
Chris B
181 ● 1 ● 8