Shahar Mike's Web Spot

shaharmike.com/cpp/move-semantics

Move Semantics

(1424 words) Wed, Sep 19, 2018
Table of Contents

Move Semantics are a C++11 feature which complements C++98's RVO; Think of them as **user-defined RVO-like optimization**. While originally designed to only allow optimizations, one can also utilize move semantics to limit APIs. This is how `std::unique_ptr` is able to be a *move-only* type, allowing it to enforce single ownership (more about `std::unique_ptr` here).

Motivation

As we saw previously, RVO does not *always* take place. When it doesn't, C++98 forced users to create expensive copies.

As an example, let's see what the following program does when compiled with different flags:

```
#include <iostream>
#include <string>

// replace operator new and delete to log allocations
void* operator new(std::size_t n) throw(std::bad_alloc) {
  std::cout << "[Allocating " << n << " bytes]\n";
  return malloc(n);
}
void operator delete(void* p) throw() { free(p); }

std::string BuildLongString() {
  return "This string is so long it can't possibly be inline (SSO)";
}

int main() {
  BuildLongString();
}
```

(1) C++98 with RVO support: Single copy - RVO FTW!

```
$ clang++-libc++ -std=c++98 main.cpp && ./a.out
[Allocating 64 bytes]
```

(2) C++98 without RVO support: 2 copies - makes sense. Sad but true.

```
$ clang++-libc++ -std=c++98 -fno-elide-constructors main.cpp && ./a.out
[Allocating 64 bytes]
[Allocating 64 bytes]
```

(3) C++11 with RVO support: Single copy - no news.

```
$ clang++-libc++ -std=c++11 main.cpp && ./a.out
[Allocating 64 bytes]
```

(4) C++11 without RVO support: Single copy - that's new!

```
$ clang++-libc++ -std=c++11 -fno-elide-constructors main.cpp && ./a.out
[Allocating 64 bytes]
```

Using move-semantics we are able to avoid allocating data even when RVO is disabled.

More Practical Example

Let's consider the following example:

```
std::string s = BuildLongString();  // Same BuildLongString() from above
// Do something with s.
s = BuildLongString();  // Copy assignment - no RVO ever!
// Do something else with s.
```

In C++98 the above code had to create a copy of the long string returned by `BuildLongString()` [1] because RVO is not allowed on assignment. That's unfortunate, because we can immediately see that the previous value of `s` will be lost as part of that assignment.

C++11's Move Semantics allow us to avoid this copy by 'stealing' the pointer of the temporary object returned by `BuildLongString()` and directing that temporary object to not own that pointer anymore (so that it won't attempt to `delete` it).

In other words, the compiler now provides us with a way to *know* that an object passed to us is temporary and will soon be destroyed. With this knowledge we can write smarter and better performing code. These temporary, soon-to-be-destroyed objects are annotated with `&&` - a new C++ syntax.

Move Constructor / Move Assignment

Most commonly, move-semantics are used for creating a special type of constructor called a *move constructor*. Move constructors are similar to *copy constructors* both syntactically and logically. They can be implemented in addition to, or instead of, a copy constructor. Similarly one can implement a *move assignment* - in addition to, or instead of a copy assignment (like in `a = b;`).

```
class MyClass {
 public:
  MyClass();                      // Constructor

  MyClass(const MyClass& o);          // Copy constructor
  MyClass(MyClass&& o);               // Move constructor

  MyClass& operator=(const MyClass& o);  // Copy assignment
  MyClass& operator=(MyClass&& o);       // Move assignment
};
```

As we saw above, `MyClass&&` is the syntax for a special reference to `MyClass` that is an rvalue, aka *rvalue reference*.

Move constructors / assignment operations will be invoked automatically by the compiler only if the parameter passed to them (`o` in the above example) are rvalues. Otherwise the compiler will invoke the safe-but-slow copy constructor / assignment.

Consider this: you're tasked with implementing `std::string` 's assignment operator. Let's assume `std::string` has 3 members: `data_` , `size_` and `capacity_` . When implementing the assignment function you obviously want `this` to be like another `std::string o` (that's

the meaning of an assignment), but you also *know* that `o` will very soon need to be destroyed. With this knowledge you can implement that assignment operation in a very optimized fashion.

```
std::string& operator=(std::string&& o) {  // `o` is a temporary
  // Steal & copy data
  data_ = o.data_;  // data_ is a char*
  size_ = o.size_;  // size_ is a size_t
  capacity_ = o.capacity_;  // capacity_ is a size_t

  // Make sure `o` can be destroyed safely
  o.data_ = nullptr;
  // We can also do o.size_ = o.capacity_ = 0;

  return *this;
}
```

No memory allocation, no copying of buffer, O(1) operation. That's much better than copy assignment!

Interim Summary

This special syntax, `std::string&& o`, is our entry point to using move semantics. Furthermore, this assignment operation is not a *copy assignment* but rather a *move assignment*. And `&&` means that we have a special reference in our hands - a reference to a "temporary object".

Temporary Objects - Intuition

What exactly is considered to be temporary?

You might have seen the term rhs (right-hand side) or rvalue (right-hand value) in some compiler errors in the past. For example, when attempting to compile code such as this:

```
int foo() { return 42; }

// ...
foo() = 5;  // Error: expression is not assignable
```

It doesn't make sense to assign *to* the value returned from `foo()`. It might have made sense if `foo()` were to return a *reference*, but that's not the case here. Since it doesn't make sense for this variable to be assigned to, the compiler forbids us to do so.

This is the first rule of thumb in deciding whether an object is a temporary: can it be used in the left-hand side of an assignment equation? That's exactly what we tried to do with `foo() = 5;` above. If we can't - the object is a temporary.

Unfortunately this rule doesn't always work. The compiler *will* allow us to invoke assignment operations on a custom class (unless they used ref-qualifiers, which are uncommon and beyond the scope of this post).

Another rule of thumb, and one I like better, is to consider whether it is possible to take the address of an object. For example:

```
int foo() { return 42; }

// ...
int i = foo();
int* p = &i;  // OK: `i` is an lvalue

p = &foo();  // Error: cannot take the address of an rvalue of type 'int'.
```

Note that it's fine to take the address of the *function*. We can't take the address of the value *returned from* a function. I.e:

```
auto t = &foo;    // OK - taking the address of function foo
auto v = &foo();  // Error - can't take the address of the value returned from foo
```

In a future post I plan to explain better what is the definition of rvalues, but for now we will consider *an object which will be destroyed by the end of the statement* as rvalue. They are usually temporary objects, as in the above. You can find a more accurate definition of what's an rvalue in cppreference.

`std::move()`

Any function (such as move constructor, move assignment, or just a global function) which accepts rvalue references (`&&`) can only be called with an rvalue object. This is where the true power of move semantics comes into play. The compiler *knows* when it's safe to pass an object as an rvalue reference. If it's not, we'll get a compile error:

```
void foo(std::string&& s) { /* ... */ }

// ...

foo("hello");  // Temporary objects are always rvalues.

// Return values are rvalues as well (except when the function returns a
// reference)
foo(BuildLongString());

std::string s;
foo(s);  // Compile error - `s` isn't an rvalue
```

This is good, and by design. However there are cases where we do want to convert an object to an rvalue - where we know it won't be used in the future. What then? This is

where `std::move()` comes into play:

```
// std::move() converts an object to rvalue.
foo(std::move(s));
```

Rule of 3 becomes Rule of 5

One last thing before I wrap up: If you ever heard of the rule of three - with move semantics we now have a complementary rule - the rule of 5.

That's it

In the next post I plan to look into what rvalue categories are and how they differ from rvalue references. See you next time!