C++11 Tutorial: Introducing the Move Constructor and the Move Assignment Operator

August 6, 2012 by Danny Kalev ▪ 13 Comments

Share

Copy constructors sounds like a topic for an article from 1989. And yet, the changes in the new C++ standard affect the design of a class' special member functions fundamentally. Find out more about the impact of move semantics on objects' behavior and learn how to implement the move constructor and the move assignment operator in C++11.

C++11 is the informal name for ISO/IEC 14882:2011, the new C++ standard that was published in September 2011. It includes the TR1 libraries and a large number of new core features (a detailed discussion about these new C++11 features is available here; also see The Biggest Changes in C++11 (and Why You Should Care)):

- Initializer lists
- Uniform initialization notation
- Lambda functions and expressions
- Strongly-typed enumerations
- Automatic type deduction in declarations
- `__thread_local` storage class
- Control and query of object alignment
- Static assertions
- Type `long long`
- Variadic templates

Important as these features may be, the defining feature of C++11 is *rvalue references*.

The Right Time for Rvalue References

Rvalue references are a new category of reference variables that can bind to *rvalues*. Rvalues are slippery entities, such as temporaries and literal values; up until now, you haven't been able to bind these safely to reference variables.

Technically, an rvalue is an unnamed value that exists only during the evaluation of an expression. For example, the following expression produces an rvalue:

```
x+(y*z); // A C++ expression that produces a temporary
```

C++ creates a temporary (an rvalue) that stores the result of `y*z`, and then adds it to `x`. Conceptually, this rvalue evaporates by the time you reach the semicolon at the end of the full expression.

A declaration of an rvalue reference looks like this:

Popular Posts

Please Stop Saying Java Sucks

14 Ways to Contribute to Open Source without Being a Programming Geniu...

Understanding SOAP and REST Basics And Differences

A Taste of Salt: Like Puppet, But Less Frustrating

There Ain't No ROI in Software Testing

C11: A New C Standard Aiming at Safer Programming

Creating your own TestSteps in soapUI

IoT is Busy Innovating, but Where's the Integration? Thoughts from O&#...

```
std::string&& rrstr; //C++11 rvalue reference variable
```

The traditional reference variables of C++ e.g.,

```
std::string& ref;
```

are now called *lvalue references*.

Rvalue references occur almost anywhere, even if you don't use them directly in your code. They affect the semantics and lifetime of objects in C++11. To see how exactly, it's time to talk about *move semantics*.

Get to Know Move Semantics

Hitherto, copying has been the only means for transferring a state from one object to another (an object's state is the collective set of its non-static data members' values). Formally, copying causes a target object `t` to end up with the same state as the source `s`, without modifying `s`. For example, when you copy a string `s1` to `s2`, the result is two identical strings with the same state as `s1`.

And yet, in many real-world scenarios, you don't copy objects but *move* them. When my landlord cashes my rent check, he moves money from my account into his. Similarly, removing the SIM card from your mobile phone and installing it in another mobile is a move operation, and so are cutting-and-pasting icons on your desktop, or borrowing a book from a library.

Notwithstanding the conceptual difference between copying and moving, there's a practical difference too: Move operations tend to be faster than copying because they transfer an existing resource to a new destination, whereas copying requires the creation of a new resource from scratch. The efficiency of moving can be witnessed among the rest in functions that return objects by value. Consider:

```
string func()
{
string s;
//do something with s
return s;
}
string mystr=func();
```

When `func()` returns, C++ constructs a temporary copy of `s` on the caller's stack memory. Next, `s` is destroyed and the temporary is used for copy-constructing `mystr`. After that, the temporary itself is destroyed. Moving achieves the same effect without so many copies and destructor calls along the way.

Moving a string is almost free; it merely assigns the values of the source's data members to the corresponding data members of the target. In contrast, copying a string requires the allocation of dynamic memory and copying the characters from the source.

Move Special Member Functions

C++11 introduces two new special member functions: the *move constructor* and the *move assignment operator*. They are an addition to the fabulous four you know so well:

- Default constructor
- Copy constructor
- Copy assignment operator
- Destructor

If a class doesn't have any user-declared special member functions (save a default constructor), C++ declares its remaining five (or six) special member functions

implicitly, including a move constructor and a move assignment operator. For example, the following class

```
class S{};
```

doesn't have any user-declared special member functions. Therefore, C++ declares all of its six special member functions implicitly. Under certain conditions, implicitly declared special member functions become implicitly defined as well. The implicitly-defined move special member functions move their sub-objects and data members in a member-wise fashion. Thus, a move constructor invokes its sub-objects' move constructors, recursively. Similarly, a move assignment operator invokes its sub-objects' move assignment operators, recursively.

What happens to a moved-from object? The state of a moved-from object is unspecified. Therefore, always assume that a moved-from object no longer owns any resources, and that its state is similar to that of an empty (as if default-constructed) object. For example, if you move a string `s1` to `s2`, after the move operation the state of `s2` is identical to that of `s1` before the move, whereas `s1` is now an empty (though valid) string object.

Designing a Move Constructor

A move constructor looks like this:

```
C::C(C&& other); //C++11 move constructor
```

It doesn't allocate new resources. Instead, it *pilfers* `other`'s resources and then sets `other` to its default-constructed state.

Let's look at a concrete example. Suppose you're designing a `MemoryPage` class that represents a memory buffer:

```
class MemoryPage
{
size_t size;
char * buf;
public:
explicit MemoryPage(int sz=512):
size(sz), buf(new char [size]) {}
~MemoryPage( delete[] buf;}
//typical C++03 copy ctor and assignment operator
MemoryPage(const MemoryPage&);
MemoryPage& operator=(const MemoryPage&);
};
```

A typical move constructor definition would look like this:

```
//C++11
MemoryPage(MemoryPage&& other): size(0), buf(nullptr)
{
// pilfer other's resource
size=other.size;
buf=other.buf;
// reset other
other.size=0;
other.buf=nullptr;
}
```

The move constructor is much faster than a copy constructor because it doesn't allocate memory nor does it copy memory buffers.

Designing a Move Assignment Operator

A move assignment operator has the following signature:

```
C& C::operator=(C&& other);//C++11 move assignment operator
```

A move assignment operator is similar to a copy constructor except that before pilfering the source object, it releases any resources that its object may own. The move assignment operator performs four logical steps:

- Release any resources that `*this` currently owns.
- Pilfer `other`'s resource.
- Set `other` to a default state.
- Return `*this`.

Here's a definition of `MemoryPage`'s move assignment operator:

```
//C++11
MemoryPage& MemoryPage::operator=(MemoryPage&& other)
{
if (this!=&other)
{
// release the current object's resources
delete[] buf;
size=0;
// pilfer other's resource
size=other.size;
buf=other.buf;
// reset other
other.size=0;
other.buf=nullptr;
}
return *this;
}
```

Overloading Functions

The overload resolution rules of C++11 were modified to support rvalue references. Standard Library functions such as `vector::push_back()` now define two overloaded versions: one that takes `const T&` for lvalue arguments as before, and a new one that takes a parameter of type `T&&` for rvalue arguments. The following program populates a vector with `MemoryPage` objects using two `push_back`() calls:

```
#include <vector>
using namespace std;
int main()
{
vector<MemoryPage> vm;
vm.push_back(MemoryPage(1024));
vm.push_back(MemoryPage(2048));
}
```

Both `push_back()` calls resolve as `push_back(T&&)` because their arguments are rvalues. `push_back(T&&)` moves the resources from the argument into `vector`'s internal `MemoryPage` objects using `MemoryPage`'s move constructor. In older versions of C++, the same program *would* generate copies of the argument since the copy constructor of `MemoryPage` would be called instead.

As I said earlier, `push_back(const T&)` is called when the argument is an lvalue:

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

```
int main()
{
vector<MemoryPage> vm;
MemoryPage mp1(1024);//lvalue
vm.push_back(mp); //push_back(const T&)
}
```

However, you can enforce the selection of `push_back(T&&)` even in this case by casting an lvalue to an rvalue reference using `static_cast`:

```
//calls push_back(T&&)
```

vm.push_back(static_cast<MemoryPage&&>(mp));

Alternatively, use the new standard function `std::move()` for the same purpose:

```
vm.push_back(std::move(mp));//calls push_back(T&&)
```

It may seem as if `push_back(T&&)` is always the best choice because it eliminates unnecessary copies. However, remember that `push_back(T&&)` empties its argument. If you want the argument to retain its state after a `push_back()` call, stick to copy semantics. Generally speaking, don't rush to throw away the copy constructor and the copy assignment operator. In some cases, the same class could be used in a context that requires pure copy semantics, whereas in other contexts move semantics would be preferable.

In Conclusion

C++11 is a different and better C++. Its rvalue references and move-oriented Standard Library eliminate many unnecessary copy operations, thereby improving performance significantly, with minimal, if any, code changes. The move constructor and the move assignment operator are the vehicles of move operations. It takes a while to internalize the principles of move semantics – and to design classes accordingly. However, the benefits are substantial. I would dare predicting that other programming languages will soon find ways to usher-in move semantics too.

Danny Kalev is a certified system analyst by the Israeli Chamber of System Analysts and software engineer specializing in C++. Kalev has written several C++ textbooks and contributes C++ content regularly on various software developers' sites. He was a member of the C++ standards committee and has a master's degree in general linguistics.

See also:

- The Biggest Changes in C++11 (and Why You Should Care)
- C11: A New C Standard Aiming at Safer Programming