



Move Semantics and Rvalue References

ITP 435 – Spring 2016

Week 5, Lecture 1

Lecturer: Sanjay Madhav



- An ***lvalue*** is a variable or object that persists beyond an expression
- An easy way to determine if something is an lvalue is whether or not you can take the address of it...

```
int a = 5;
```

```
&a; // valid, because a is an lvalue
```

```
&(a + 1); // invalid, because (a + 1) is not an lvalue
```

```
int* p1 = &a;
```

```
&p1; // valid, because p1 is an lvalue
```

```
&(++p1); // valid, because preincrement modifies lvalue
```

```
&(p1++); // invalid, because p1++ creates copy that's not an lvalue
```

- Error message is: error C2102: '&' requires l-value

Lvalues, cont'd



- Lvalues also can appear on the left side of an assignment, like:

```
int i, j, *p;
```

```
// Correct usage: the variable i is an lvalue.
```

```
i = 7;
```

```
// Incorrect usage: The left operand must be an lvalue (C2106).
```

```
7 = i; // C2106
```

```
j * 4 = 7; // C2106
```

```
// Correct usage: the dereferenced pointer is an lvalue.
```

```
*p = i;
```

```
// Correct usage: the conditional operator returns an lvalue.
```

```
((i < 3) ? i : j) = 7;
```

Example code from Visual Studio documentation:
<http://msdn.microsoft.com/en-us/library/f90831hc.aspx>

References and lvalues



- A normal reference (herein called an *lvalue reference*) can only refer to an lvalue

- That's because references work based on memory addresses!

```
int a = 5;
```

```
int& ref = a; // valid, because a is an lvalue
```

```
int& ref2 = (a + 1); // invalid -- (a + 1) is not an lvalue
```

- Error: C2440: 'initializing' : cannot convert from 'int' to 'int &'



- The opposite of an lvalue is an *rvalue*
- Any hidden variables that are created as a result of expressions or function calls are rvalues:

```
int a = 5;  
&a; // a is an lvalue  
&(a + 1); // invalid, because (a + 1) is an rvalue  
  
int* p1 = &a;  
&p1; // valid, because p1 is an lvalue  
&(++p1); // valid, because preincrement modifies lvalue  
&(p1++); // invalid, because p1++ creates an rvalue copy
```

More Examples



```
// lvalues:
int i = 42;
i = 43; // ok, i is an lvalue
int* p = &i; // ok, i is an lvalue
int& foo();
foo() = 42; // ok, foo() is an lvalue
int* p1 = &foo(); // ok, foo() is an lvalue

// rvalues:
int foobar();
int j = 0;
j = foobar(); // ok, foobar() is an rvalue
int* p2 = &foobar(); // error, cannot take the address of an rvalue
j = 42; // ok, 42 is an rvalue
```

From: http://thbecker.net/articles/rvalue_references/section_01.html

Simple String Class



```
#include <cstring>
class string
{
    char* m_str;
    size_t m_len;
public:
    // Assume we have regular constructor, destructor, and
    // operator+ defined also
    string(const string& rhs) // copy constructor
    {
        m_len = rhs.m_len;
        m_str = new char[m_len + 1];
        memcpy(m_str, rhs.m_str, m_len + 1);
    }
};
```

Unnecessary Copying



- Suppose you have the following strings:

```
string a("1234");
```

```
string b("5678");
```

- What happens when you do this?

```
string c(a + b);
```

1. operator+ will construct a new string ("12345678") which it returns by *value*
2. c will then call the copy constructor which will allocate more memory and then copies "12345678" into it

This wastes memory!!!

The string in step 1 is an rvalue, so we should steal it's m_str

Move Constructor



- Relies on && which is an *rvalue reference*

```
// Add this "move constructor" to string class
// Steal the rvalue's data!!!
string(string&& rvalue)
{
    m_str = rvalue.m_str;
    m_len = rvalue.m_len;

    // Null so rvalue destructor won't delete the data
    rvalue.m_str = nullptr;
}
```

Try this again



- Suppose you have the following strings:
`string a("1234");`
`string b("5678");`
 - Now that we have a move constructor, what happens here?
`string c(a + b);`
1. `operator+` will construct a new string ("12345678") which it returns by *value*
 2. `c` will then call the move constructor, which will steal the data from the `(a + b)` rvalue
- **Success!!**
 - **Well, sort of...**

Another Example...



```
struct Test {  
    // Default constructor  
    Test() {  
        std::cout << "Default" << std::endl;  
        mValue = 0;  
    }  
    // Copy constructor  
    Test(const Test& rhs) {  
        std::cout << "Copy" << std::endl;  
        mName = rhs.mName;  
        mValue = rhs.mValue;  
    }  
    // Move constructor  
    Test(Test&& rhs) {  
        std::cout << "Move" << std::endl;  
        mName = rhs.mName;  
        mValue = rhs.mValue;  
    }  
  
    std::string mName;  
    int mValue;  
};
```

Then I use it...



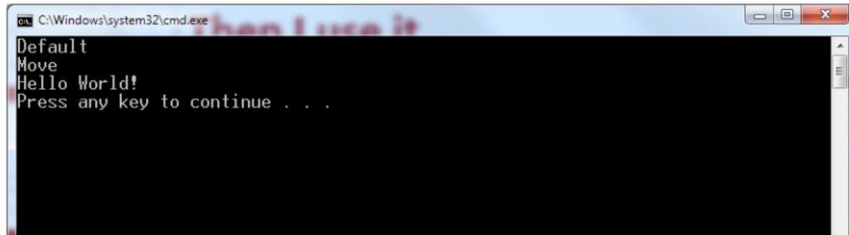
- Then if I have this:

```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}  
  
int main() {  
    Test a(doStuff());  
    std::cout << a.mName << std::endl;  
    return 0;  
}
```

Output



- Here's the output...it seems to work



```
C:\Windows\system32\cmd.exe
Default
Move
Hello World!
Press any key to continue . . .
```

- But there's a problem!

What's wrong with this move constructor?



```
// Move constructor
Test(Test&& rhs) {
    std::cout << "Move" << std::endl;
    mName = rhs.mName;
    mValue = rhs.mValue;
}
```

What happens
with the string
member?

What's wrong with this move constructor?



```
// Move constructor
Test(Test&& rhs) {
    std::cout << "Move" << std::endl;
    mName = rhs.mName;
    mValue = rhs.mValue;
}
```

First mName will
be constructed w/
default
constructor...

Then we call the
assignment
operator...

First, we should use object initializer syntax!



```
struct Test {  
    // Default Constructor  
    Test()  
        : mValue(0)  
    { std::cout << "Default" << std::endl; }  
  
    // Copy constructor  
    Test(const Test& rhs)  
        : mName(rhs.mName)  
        , mValue(rhs.mValue)  
    { std::cout << "Copy" << std::endl; }  
  
    // Move constructor  
    Test(Test&& rhs)  
        : mName(rhs.mName)  
        , mValue(rhs.mValue)  
    { std::cout << "Move" << std::endl; }  
  
    std::string mName;  
    int mValue;  
};
```




- Now what happens?

```
// Move constructor
Test(Test&& rhs)
: mName(rhs.mName)
, mValue(rhs.mValue)
{
    std::cout << "Move" << std::endl;
}
```

mName is
constructor with
rhs.mName,
which calls which
constructor?

Copy constructor (not move!)



- Defined in `<utility>`
- `std::move` casts from an lvalue to an rvalue reference:

```
Test(Test&& rhs)
: mName(std::move(rhs.mName))
, mValue(std::move(rhs.mValue))
{
    std::cout << "Move" << std::endl;
}
```

- This way, `mName` will be constructed with a move constructor, if one exists for `std::string` (which it does!)



- When we do a `std::move` we actually get an xvalue.
- From the C++ standard:
“An **xvalue** is an expression that identifies an "eXpiring" object, that is, the object that may be moved from. The object identified by an xvalue expression may be a nameless temporary, it may be a named object in scope, or any other kind of object, but if used as a function argument, xvalue will always bind to the rvalue reference overload if available”
- You can think of an xvalue as a subset of rvalue (though technically it isn't)

The “Rule of three”



The “[Rule of three](#)” in C++ states that if you are compelled to implement any of the following three class members:

- Destructor
- Copy Constructor
- Assignment Operator

...you should most likely implement all three of them – otherwise bad things can happen!

So for instance, if you are dynamically allocating stuff, that tells you that you need a destructor to prevent memory leaks. By the rule of three, this means you probably need copy constructor and assignment to ensure you are doing deep copies.

The “Rule of three five”



The “[Rule of three five](#)” in C++ states that if you are compelled to implement any of the following three class members:

- Destructor
- Copy Constructor
- Assignment Operator
- **Move Constructor**
- **Move Assignment Operator**

...you should most likely implement all five of them – otherwise bad things can happen!

So for instance, if you are dynamically allocating stuff, that tells you that you need a destructor to prevent memory leaks. By the rule of three, this means you probably need copy constructor and assignment to ensure you are doing deep copies.

The “Rule of zero”



There's also the “rule of zero” which says that in modern C++, you shouldn't have to overload any of the five member functions!

This can only be the case if you avoid using `new` altogether and instead use:

- STL collections
- Smart pointers

Destructor, Assignment and Move Assignment



```
// Destructor
~Test() { std::cout << "Destructor" << std::endl; }

// Assignment
Test& operator=(const Test& rhs) {
    std::cout << "Assignment" << std::endl;
    mName = rhs.mName;
    mValue = rhs.mValue;
    return *this;
}

// Move Assignment
Test& operator=(Test&& rhs) {
    std::cout << "Move Assignment" << std::endl;
    mName = std::move(rhs.mName);
    mValue = std::move(rhs.mValue);
    return *this;
}
```

Notice how we use `std::move` in move assignment to try to allow for the move assignment operator to also happen for member variables

Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}  
  
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```


Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}  
  
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```

Output

Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}  
  
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```

Output

1. Default

Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}
```

Output

1. Default
2. Default

```
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```

Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}  
  
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```

Output

- | |
|------------|
| 1. Default |
| 2. Default |
| 3. Move |

Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}  
  
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```

Output

- | |
|---------------|
| 1. Default |
| 2. Default |
| 3. Move |
| 4. Destructor |

Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}  
  
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```

Output

1. Default
2. Default
3. Move
4. Destructor
5. Move Assignment

Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}
```

```
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```

Output

1. Default
2. Default
3. Move
4. Destructor
5. Move Assignment
6. Destructor
7. Hello World!

Another Test...



```
Test doStuff() {  
    Test temp;  
    temp.mName = "Hello World!";  
    return temp;  
}  
  
int main() {  
    Test b;  
    b.mName = "Goodbye!";  
    b = doStuff();  
    std::cout << b.mName << std::endl;  
    return 0;  
}
```

Output

1. Default
2. Default
3. Move
4. Destructor
5. Move Assignment
6. Destructor
7. Hello World!
8. Destructor

A quote from Effective Modern C++ (Item #29)



“Move semantics is arguably *the* premier feature of C++11. ‘Moving containers is now as cheap as copying pointers!’ you’re likely to hear, and ‘Copying temporary objects is now so efficient, coding to avoid it is tantamount to premature optimization!’ Such sentiments are easy to understand. Move semantics is truly an important feature. It doesn’t just allow compilers to replace expensive copy operations with comparatively cheap moves, it actually *requires* that they do so (when the proper conditions are fulfilled). Take your C++98 code base, recompile with a C++11-conformant compiler and Standard Library, and—*shazam!*—your software runs faster.”

More From Item #29...



“There are thus several scenarios in which C++11’s move semantics do you no good:

- **No move operations:** The object to be moved from fails to offer move operations. The move request therefore becomes a copy request.
- **Move not faster:** The object to be moved from has move operations that are no faster than its copy operations.
- **Move not usable:** The context in which the moving would take place requires a move operation that emits no exceptions, but that operation isn’t declared noexcept.

It’s worth mentioning, too, another scenario where move semantics offers no efficiency gain:

- **Source object is lvalue:** With very few exceptions (see e.g., Item 25) only rvalues may be used as the source of a move operation.”