# *rvalue* references
# Move Semantics
## *rvalues* and *lvalues*

Michael R. Nowak
Texas A&M University

# What's this error even mean?

- Not something you'd write, but consider the error message produced by

```
int two_back ()
{
    return 2;
}

int main (int argc, char * argv[])
{
    two_back() = 2;
    return 0;
}
```

```
Test.cpp: In function 'int main(int, char**)':
Test.cpp:18:18: error: lvalue required as left operand of assignment
     two_back() = 2;
                  ^
```

What's an *lvalue*?

# What's this error even mean?

- Again, not something you'd write, but consider the error message produced by (note: return type changed to `int&`)

```
int& two_back ()
{
    return 2;
}
```

```
int main (int argc, char * argv[])
{
    two_back() = 2;
    return 0;
}
```

What's an *rvalue*?

```
Test.cpp: In function 'int& two_back()':
Test.cpp:7:12: error: invalid initialization of non-const reference of
type 'int&' from an rvalue of type 'int'
     return 2;
            ^
```

# lvalues and rvalues

- An *lvalue* represents an object that occupies an identifiable location in memory that lives beyond an expression

```
int i;
i= 4;
```

An assignment expects an *lvalue* as its left operand; `i` is an *lvalue*: it is an object with an identifiable memory location

- A *rvalue* is defined by exclusion, by saying that every expression is either an *lvalue* or an *rvalue*, and that an *rvalue* is not an *lvalue*

```
(i % 2) = 0; // Error
```

The expression `i % 2` produces an *rvalue*; that is, a temporary result of an expression; the resulting *rvalue* does not live beyond the expression that produces it

# What's this error even mean?

- Not something you'd write, but consider the error message produced by

```
int two_back ()
{
    return 2;
}

int main (int argc, char * argv[])
{
    two_back() = 2;
    return 0;
}
```

```
Test.cpp: In function 'int main(int, char**)':
Test.cpp:18:18: error: lvalue required as left operand of assignment
     two_back() = 2;
                  ^
```

We attempted to assign to an $rvalue$ (here, the result of an expression that is not an $lvalue$); the compiler expected an $lvalue$

# What's this error even mean?

- Again, not something you'd write, but consider the error message produced by (note: return type changed to `int&`)

```
int& two_back ()
{
    return 2;
}
```

```
int main (int argc, char * argv[])
{
    two_back() = 2;
    return 0;
}
```

A reference is an alias to a variable; we cannot create one to an integer literal because it is an $rvalue$ – it does not have an identifiable location that we are aware of

```
Test.cpp: In function 'int& two_back()':
Test.cpp:7:12: error: invalid initialization of non-const reference of
type 'int&' from an rvalue of type 'int'
     return 2;
            ^
```

# lvalues and rvalues

- One of the main differences between $lvalues$ and $rvalues$ is that $lvalues$ can be modified
- C++11 has introduced the ability to, in special circumstances, have a reference to an $rvalue$ and thus modify them

# References to rvalues

- The ability to create references to $rvalues$ under special circumstances has powerful implications
- We will see why in a moment when we begin talking about move semantics

# But first...

- Let's update our MyLinkedList class to include print statements in each of the constructors
- Then create two instances of MyLinkedList
- And observe how the constructors are called during the assignment of one to the other

- We'll then build up to how $rvalue$ references and the related concept move semantics is so powerful

# Creating two linked lists

- Let's create a MyLinkedList l1{29} and a second MyLinkList l2{34}; here's how they look (along the constructor calls)

```
[0x7fff50c58370] Constructor(int) called
MyLinkedList l1
.----.
|Head|
 0x7fdf89c04c50
.----.
| 29 |
'----'
|Tail|
'----'
```

```
[0x7fff50c58360] Constructor(int) called
MyLinkedList l2
.----.
|Head|
 0x7fdf89c04c80
.----.
| 34 |
'----'
|Tail|
'----'
```

# Assigning an lvalue

- To see how the constructor calls are made during an assignment of an lvalues to an lvalue, we assigned l1 to l2
- The constructors that were called for this process reflect how my operator= was implemented for this class

```
assigning lvalue...
[0x7fff50c58360] Copy assignment operator called
[0x7fff50c58320] Copy constructor called
ending assignment
```

  - I updated how this was accomplished since our last lecture to use the *copy-and-swap idiom*, which is beyond the scope of this course, but wanted y'all to have code that uses
  - The *copy-and-swap idiom* is a pretty cool way to take advantage of the copy constructor to implement the copy assignment operator, which is why you'll see the copy copy constructor when you would have expected only the copy assignment operator

# Assigning an rvalue

- To see how the constructor calls are made during an assignment of an rvalue to an lvalue, we assigned MyLinkedList(2) to l2

```
l2 = MyLinkedList(2);
```

- Note that a temporary object of MyLinkedList(2) is created first, which is then assigned to l2

```
[0x7fff50c58380] Constructor(int) called
[0x7fff50c58360] Copy assignment operator called
[0x7fff50c58320] Copy constructor called
```

# Assigning an rvalue

- For the assignment to occur, a temporary object (i.e., the rvalue) had to be built first, only to be destroyed moments later

- That doesn't sound very efficient, especially if the building of these objects is computationally expensive

# rvalue references

- When temporary objects like that discussed on the previous slide are constructed, C++ allows us to create a reference to them
- The syntax for declaring an $rvalue\ reference$ to a temporary object of our MyLinkedList class is

```
MyLinkedList&& identifier_name
```

- However, recall that $rvalues$ do not live past the expression in which they arise
- So... how can a reference to an $rvalue$ be useful then?

# Move semantics

- We can implement an additional constructor and another overloaded operator= for our MyLinkedList class that take $rvalue$ $references$ as arguments
  - Then when an $rvalue$ is then used to initialize, or during the assignment, of a MyLinkedList object, the constructor, or overloaded operator=, taking an $rvalue$ argument will be called

# Move semantics

- We *can* then *pilfer/steal* the *dynamic parts from* the *temporary object* (i.e., the *rvalue*) and *use* them for those of the *object* being *initialized* or getting *assigned*
  - That is, we can use what is already there and would be destructed anyway
  - Thus avoiding the need to perform a potentially expensive copy of the data

# MyLinkedList : Move Constructor Declaration

```
MyLinkedList(MyLinkedList&& source);
```

&& indicates that it is an Rvalue,
i.e. a temporary object.

# MyLinkedList : Move Constructor Definition

```
MyLinkedList::MyLinkedList(MyLinkedList&& source) {
      // pilfer dynamic resources from source
      head = source.get_head();
      // set dynamic resources to nullptr
      source.null_head(true);
}
```

More details here:
https://msdn.microsoft.com/en-us/library/dd293665.aspx

# Move Assignment Declaration

```
MyLinkedList& operator=(MyLinkedList&& source);
```

&& indicates that it is an Rvalue,
i.e. a temporary object.

# Move Assignment Definition

```
MyLinkedList& operator=(MyLinkedList&& source) {
      if (this != &source) {
            // delete old data (not shown)
            // pilfer resources from source
            head = source.get_head();
            // set pointers in source to nullptr
            source.null_head(true);
      }
      return *this;
}
```

# Assigning an rvalue and move semantics

- To see how the constructor calls are made during an assignment of an rvalue to an lvalue once we've implemented move semantics, we assigned MyLinkedList(22) to a MyLinkedList object

```
MyLinkedList l3{11};
l3 = MyLinkedList(22);
```

- We observe two constructor calls, one for the temporary object and the other to the move assignment operator

# Assigning an *rvalue* and move semantics

MyLinkedList l3{11} initialization

```
[0x7fff50580370] Constructor(int) called
l3
.----.
|Head|
 0x7f89f2c04c50
.----.
| 11 |
'----'
|Tail|
'----'
```

l3 = MyLinkedList{22} (assignment of rvalue)

```
assigning rvalue...
[0x7fff50580380] Constructor(int) called
[0x7fff50580370] Move asssignment operator called

 rvalue
 .----.
 |Head|
 0x7f89f2d00000
 .----.
 | 22 |
 '----'
 |Tail|
 '----'
```

State of l3 after move assignment operator

```
.----.
|Head|
0x7f89f2d00000
.----.
| 22 |
'----'
|Tail|
'----'
```

# Recall

**Move ~~Copy~~ Assignment**

1. Check for self-assignment
2. Delete old data
3. ~~Allocate new memory~~
4. ~~Copy data from source~~
3. Pilfer resources from source
4. Set pointers in source to nullptr

**Move ~~Copy~~ Constructor**

1. ~~Allocate new memory~~
2. ~~Copy data from source~~
1. Pilfer resources from source
2. Set pointers in source to nullptr

# References


- http://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c/
- http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html
- https://msdn.microsoft.com/en-us/library/dd293665.aspx
- https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)