CS 105C: Lecture 8

Last Time...

Iterators

A way to abstract out "go through every element in the collection."

Can be thought of as a "superpowered pointer," implemented by a class

Have different capabilities: **r**eading, w**r**iting, and a**r**ithmetic

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
all categories				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
Random Access	Bidirectional	Forward	Input	Can be dereferenced as an *rvalue*	`*a` `a->m`
Random Access	Bidirectional	Forward	Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
Random Access	Bidirectional	Forward		*default-constructible*	`X a;` `X()`
Random Access	Bidirectional	Forward		Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

Random Access	Bidirectional			Can be decremented	`--a`
					`a--`
					`*a--`

Random Access				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

Random Access				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

Random Access				Supports compound assignment operations += and -=	`a += n` `a -= n`
Random Access				Supports offset dereference operator (`[]`)	`a[n]`

Consists of common functions

A first valid element	begin()
A current element	(no special method)
A first invalid element	end()
A way to access the data within the element	* [dereference]
A way to get the next element	next() or ++

An associated data structure

Lambda functions

Anonymous functions that can be declared locally. Have three parts:

```
[=](int x, int y) -> bool { return x <= y; }
```

The *capture block* | Parameters and return type | The *function body*

How can this compile?

With a lot of difficulty. But it turns out that this pattern is unambiguous in C++.

Rules for reading <algorithm> documentation

- **Rule 1**: You can pretend the ExecutionPolicy overloads don't exist
- **Rule 2**: Look at the types and names in the *simplest* signature and think about what they mean.
- **Rule 3**: Any unpaired iterators (e.g. d_first, first2) are assumed to point to a range large enough to be appropriate for the first range.
- **Rule 4**: **Unary** lambdas take one operand, **Binary** lambdas take two. **Predicate**s return booleans, and **Op**s return anything.

std::unique_copy

Defined in header <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first );                                  (1) (until C++20)
template< class InputIt, class OutputIt >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first );                            (since C++20)
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2 >
ForwardIt2 unique_copy( ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last,
                        ForwardIt2 d_first );                              (2) (since C++17)
template< class InputIt, class OutputIt, class BinaryPredicate >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first, BinaryPredicate p );               (3) (until C++20)
template< class InputIt, class OutputIt, class BinaryPredicate >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first, BinaryPredicate p );         (since C++20)
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt2 unique_copy( ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last,
                        ForwardIt2 d_first, BinaryPredicate p );           (4) (since C++17)
```

Questions!

Q: What happens if you change the captured variable?

```cpp
int main(){
    int x = 2;
    auto add_x = [ x](int z){return x + z;};
    x = 5;
    std::cout << add_x(3) << std::endl;
}
```

```cpp
int main(){
    int x = 2;
    auto add_x = [&x](int z){return x + z;};
    x = 5;
    std::cout << add_x(3) << std::endl;
}
```

```
~/t/C++ took 3s
❯ g++ lambda.cpp

~/t/C++
❯ ./a.out
5
```

```
~/t/C++
❯ g++ lambda.cpp

~/t/C++
❯ ./a.out
8
```

Q: How do I write a custom iterator for my class?

A1: For full details, see
https://users.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterator-define.html

A2: You need to write your own iterator class. It'll need to implement at least the following custom ops:

- ***** (dereference operator)
- **++** (increment operator)
- **==** and **!=** (equality test operators)

Ideally, you also modify your class to provide the begin() and end() methods, which return iterators.

CS 105C: Lecture 8

LVals and Rvals and move (oh my!)

Warning: This is the **most advanced subject** we've encountered so far (possibly on-par with templates), and dives deep into the innards of C++.

This presentation has been kept deliberately short: ask lots of questions!

Copy Constructors

and

Copy Assignment

Copies

```
int x = 3;
int y = x;

y = 5;

std::cout << x << ", " << y << std::endl;
```

What does this print and why?

Copy constructors and copy assignment operators let us customize the behavior of assignment for our classes.

```
int main(){
  Dog thalia;
  Dog buck;

  Dog dog2 = thalia;
  buck = dog2;
  dog2.say_name();
}

class Dog {
   std::string name;

   Dog(const& Dog other){
     this->name = other.name;
   }

   Dog& operator=(const Dog& other){
     this->name = other.name;
     return *this;
   }

   void say_name(){
     std::cout << "Woof I am " << name << std::endl;
   }
};
```

Calls custom copy ctor

Calls custom assignment

Sometimes, copies are expensive!

```
class IntVector {
  int* data;

  IntVector(const IntVector& other){
    for(size_t i = 0; i < other.size(); i++){
      data[i] = other[i];
    }
  }

  IntVector& operator=(const Dog& other){
    for(size_t i = 0; i < other.size(); i++){
      data[i] = other[i];
    }
    return *this;
  }
};

int main(){
  IntVector a = /* some initialization function */;
  IntVector b;
  b = a;   // How long does this take?
}
```

Consider the following code:

```
class X{
  int* data;
 public:
  X(){ expensive_operation1(); }
  X& operator=(const X& other){
    expensive_copy_operations(other);
  }
  ~X(){ expensive_operation2();}
};

X create_an_x(){
  X x;
  expensive_operation_3(x);
  return x;
}

int main(){
  X x;
  ...
  x = create_an_x();
}
```

In the absence of compiler optimizations, how many expensive operations are executed?

Consider the following code:

```
class X{
  int* data;
  X(){ expensive_operation1(); }
  X(const X& other){
     expensive_copy_operations();
  }
  X& operator=(const X& other){
     expensive_copy_operations();
  }
  ~X(){ expensive_operation2();}
};

X create_an_x(){
  X x;
  expensive_operation_3(x);
  return x;
}

int main(){
  X x;
  ...
  x = create_an_x();
}
```

1: Construction

2: Temp Obj Construction

3: Copy assignment from temporary

4: Destruct the temporary

Consider the following code:

```
class X{
  int* data;
  X(){ expensive_operation1(); }
  X& operator=(const X& other){
     expensive_copy_operations();
  }
  ~X(){ expensive_operation2();}
};

X create_an_x(){
  X x;
  expensive_operation_3(x);
  return x;
}

int main(){
  X x;
  ...
  x = create_an_x();
}
```

In this specific case, the compiler can take advantage of the *return value optimization* to avoid making copies--but this isn't always possible!

Assuming that construction, copy, and destruction are all expensive operations, **how many expensive operations are requested on line 18?**

```
1  X create_an_x(int i){
2    X x;
3    expensive_operation_3(x, i);  // Assume no copy made here
4    return x;
5  }
6
7  X process_x(X x_in){
8    X x = x_in;
9    expensive_operation_z(x); // Assume no copy made here
10   return x;
11 }
12
13 int main(){
14   X x;
15   std::vector<X> xs;
16   ...
17   for(int i = 0; i < BIG_NUMBAH; i++){
18     xs.push_back(process_x(create_an_x(i)));
19   }
20 }
```

Even worse: swapping!!

```cpp
template <typename T>
T swap(T& a, T& b){
    T temp = b;
    b = a;
    a = temp;
}
```

If T is std::vector<int> and the two inputs are each 100,000 elements large, we need to:

- Copy 800kB of memory from b to temp
- Copy 800kB of memory from a to b
- Copy 800kB of memory from temp to a
- Destroy temp

Total amount of memory written: 2.4 MB

Optimal swap algorithm writes 24 bytes of memory!

What we'd really like to have:

Could we just...steal the data instead of making an expensive copy?

```
class X{
  int* data;
  X(){ expensive_operation1(); }
  X& operator=(const X& other){
    // Yoink! Data is mine now!
    std::swap(other.data, this->data);
  }
  ~X(){ expensive_operation2();}
};

X create_an_x(){
  X x;
  expensive_operation_3(x);
  return x;
}

int main(){
  X x;
  ...
  x = create_an_x();
}
```

We *know* that we aren't going to use the RHS of this again!!

So just swap the data pointers instead of mucking around with copies!

Could we just...steal the data instead of making an expensive copy?

Answer: nope.

```
class X{
  int* data;
  X(){ expensive_operation1(); }
  X& operator=(const X& other){
    // Yoink! Data is mine now!
    std::swap(other.data, this->data);
  }
  ~X(){ expensive_operation2();}
};

X create_an_x(){
  ...
}

int main(){
  X x, x2;
  ...
  x2 = create_an_x();
  x = x2;
}
```

C++ rules say that x should be a copy of x2 here--swapping their data is going to be very, very confusing!

```
class X{
  int* data;
  X(){ expensive_operation1(); }
  X(const X& other){
    // Yoink! Data is mine now!
    std::swap(other.data, this->data);
  }
  ~X(){ expensive_operation2();}
};

X create_an_x(){
  ...
}

int main(){
  X x, x2;
  ...
  x2 = create_an_x();
  x = x2;
}
```

Wait a minute...

Is it okay for us to steal the result of create_an_x()?

What makes it different from stealing the value of x2?

LValues and RValues

In C++, some things can go on the left side, and some things can go on the right side.

```
x = 5;   // Okay!
y = 5;   // Also okay!
5 = y;   // Not okay!
x*y = 5; // Also not okay!
```

Rough intuition: **named** locations in memory can be treated as lvalues. Everything else is an rvalue.

RValues **must** go on the right hand side of an assignment operation. **Only** lvalues can appear on the left hand side of assignment.

References are restricted!

```
int& x = 5; // This is not legal!

/////////////////////////////////
int x = 5;
int& x2 = x;  // This is fine!
```

Why is the first line illegal?

In general, you may not take a non-const reference to an rvalue, because there may be no memory location to modify!

```
int& x = 3;

x++;  // What the heck does this modify? the literal value 3?

/////////////////////////////////////

const int& x = 3;  // This is okay
```

Taking const references to rvalues is okay: we can't modify them.

Can't take non-const reference to rvalue

```
int test_ref(const int& x){
  return x + 2;
}

int main(){
  int c = test_ref(3);
}
```

Compiles fine!

```
int test_ref(int& x){
  return x + 2;
}

int main(){
  int c = test_ref(3);
}
```

error: cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'

Introducing: RValue References!

We can now bind a reference to rvalues!

```cpp
int&& x = 5;
```

To avoid confusion, the old reference type is now called an "lvalue reference".

```cpp
int x = 3;       // Value
int&& x1 = 5;    // Rvalue reference
int&  x2 = x;    // Lvalue reference
```

Introducing: RValue References!

We can now bind a reference to rvalues!

```
int&& x = 5;
```

To avoid confusion, the old reference type is now called an "lvalue reference".

```
int x = 3;        // Value
int&& x1 = 5;     // Rvalue reference
int&  x2 = x;     // Lvalue reference
```

Note: Rvalue references will only bind to rvalues!!

Introducing: RValue References!

Can I bind a... \ ...to a	Lvalue Reference	const Lvalue Reference	Rvalue Reference
Lvalue	`int y;` `int &x = y;`	`int y;` `const int &x = y;`	`int y;` `int &&x = y;`
Rvalue	`int &x = 5;`	`const int &x = 5;`	`int &&x = 5;`

What can we do with rvalue references?

Binding an rvalue reference to a temporary extends its lifetime

```cpp
int main(){
  get_best_dog();
  ...
  ...
}
```

Temporary destructed here

```cpp
int main(){
  Dog&& dog = get_best_dog();
  ...
  ...
}
```

Temporary destructed here

What can we do with rvalue references?

Modify temporary values (don't worry, the compiler makes a copy before you do this!)

```
int&& x = 5;
x = x + 5;
std::cout << x; // Prints 10
```

What can we do with rvalue references?

Overload functions! Rvalues can bind to both rvalue and const lvalue references, but will preferentially select the rvalue overload if it exists.

```
void derp(const int& x){
  std::cout << "I have an lvalue!" << std::endl;
}

void derp(int&& x){
  std::cout << "I have an rvalue!" << std::endl;
}

int main(){
  int x = 5;
  derp(x);
  derp(5);
}
```

RValue Reference Overloads

(i.e. "Move Semantics")

The big thing about rvalue references isn't how you *use* them in code, it's for *overloading* functions.

Specifically, the constructor and assignment operator.

```
// Expensive to Copy!
struct ETC{
  int* data;
  int size;
  ETC();
  ETC(const ETC& other);
  ETC& operator=(const ETC& other);
  ETC(ETC&& other);
  ETC& operator=(ETC&& other);
};
```

- `ETC();` ← Default Constructor
- `ETC(const ETC& other);` ← Copy Constructor
- `ETC& operator=(const ETC& other);` ← Copy Assignment Operator
- `ETC(ETC&& other);` ← Move Constructor
- `ETC& operator=(ETC&& other);` ← Move Assignment Operator

```
X create_an_x(){
  X x;
  expensive_operation_3(x);
  return x;
}

int main(){
  X x;
  X x2;
  ...
  x2 = create_an_x();
  x = x2;
}
```

It is okay to steal this object's data in the assignment......because this is an *rvalue!*

...but not this object's data....because this is an *lvalue!*

```cpp
// Expensive to Copy!
struct ETC{
  int* data;
  int size;
  ETC() = something;
  ETC(const ETC& other) = something;
  ETC& operator=(const ETC& other) = something;
  ETC(ETC&& other) noexcept : data{nullptr}, size{0} {
    std::swap(data, other.data);
    std::swap(size, other.size);
  }
  ETC& operator=(ETC&& other) noexcept {
    std::swap(data, other.data);
    std::swap(size, other.size);
  }
};
```

```cpp
1 ETC generate_ETC(){
2   return ETC();
3 }
4
5 int main(){
6   ETC a;
7   ETC b = a;
8   ETC c = generate_ETC();
9 }
```

```cpp
// Expensive to Copy!
struct ETC{
  int* data;
  int size;
  ETC();
  ETC(const ETC& other);
  ETC& operator=(const ETC& other);
  ETC(ETC&& other);
  ETC& operator=(ETC&& other);
};
```

- Line 6: Calls default constructor
- Line 7: Calls copy constructor
- Line 8: Calls move constructor

```cpp
int main(){
  X x;
  std::vector<X> xs;
  ...
  for(int i = 0; i < BIG_NUMBAH; i++){
    xs.push_back(process_x(create_an_x(i)));
  }
}
```

Moves can be chained!

std::move

Like it's cousin remove_if, move is confusingly named because **it doesn't actually move anything!!**

```
int main(){
  ETC a;
  ETC b = std::move(a);
}
```

std::move converts its argument into an rvalue reference-to-object, allowing you to use the move constructor.

After being moved-from, *a* is in an unknown state--it is the programmer's responsibility not to rely on anything about the value of *a*.

Rule of Five

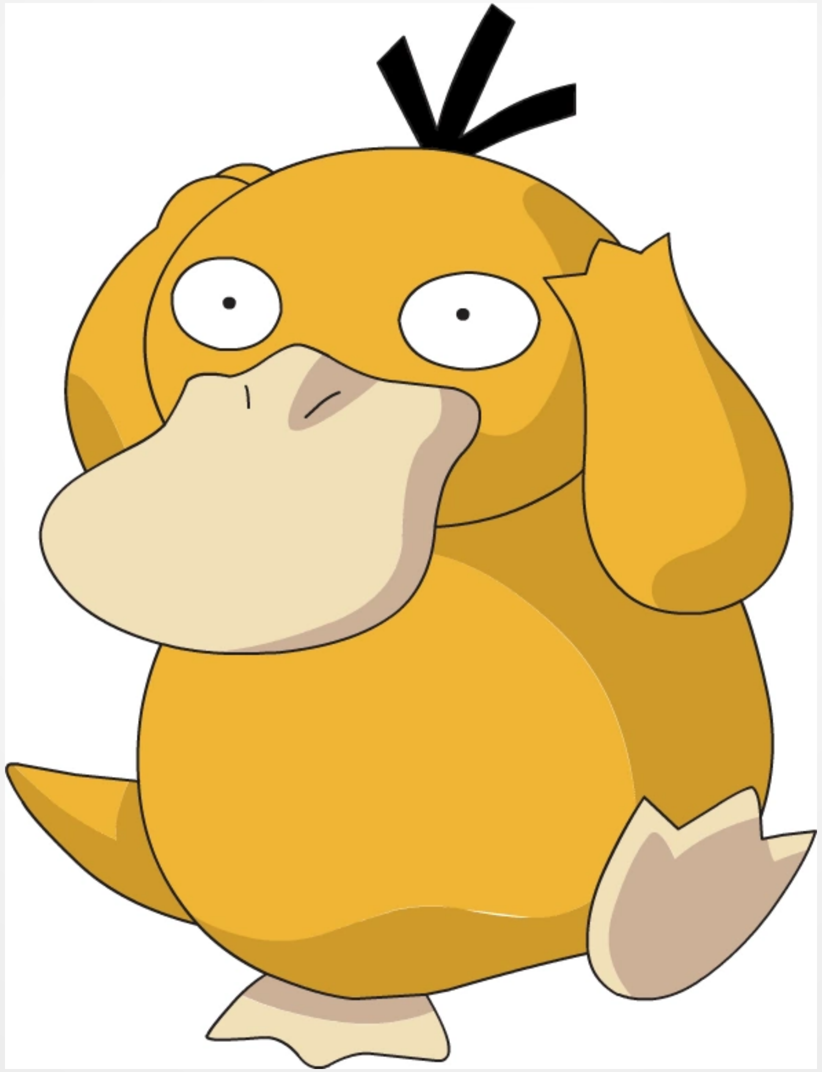
If your class implements a non-default version of any of the following functions:

- Destructor
- Copy Constructor
- Copy Assignment
- Move Constructor
- Move Assignment

then it *almost certainly* needs a custom version of all five of them.

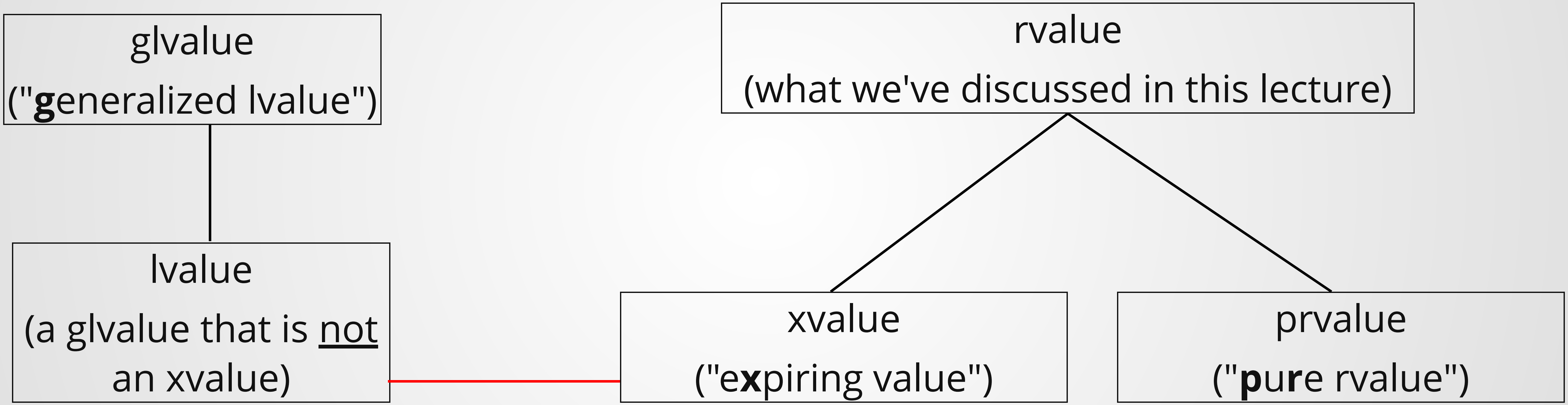
Another way of saying this is "if you define or =delete any default operation, define or =delete all of them."

Some Confusing Points



lvalues and rvalues are a simplification!

The C++ standard actually defines **five** distinct value categories!



You do **not** need to memorize this information! Just remember the names in case you run across them in the future.

```cpp
struct Tester{
  Tester(){
    std::cout << "Default constructor called!\n";
  }
  Tester(const Tester& other){
    std::cout << "Copy constructor called!\n";
  }
  Tester(Tester&& other){
    std::cout << "Move constructor called!\n";
  }
};
```

```cpp
Tester gen_tester() {
  return Tester();
}

int main(){
  Tester&& a = gen_tester();
  std::cout << "NEXT!" << std::endl;
  Tester b = a;
}
```

```
~/tmp
❯ g++ move.cpp -std=c++11 -fno-elide-constructors -o move

~/tmp
❯ ./a.out
Default constructor called!
Move constructor called!
NEXT!
Copy constructor called!
```

Rvalue references are lvalues!!

If you think about this carefully, it's actually not terribly surprising:

- Rvalue references are a named memory location
- We use rvalue capture to indicate that something is a temporary that nobody else can access--if you bind an rvalue to an rvalue reference, this is no longer true.

...but it *will* catch you off guard a few times.

Summary

Copying is expensive, stealing is cheap!

```
int main(){
  X x, x2;
  ...
  x2 = create_an_x();
  x = x2;
}
```

Wherever possible, we'd like to move data around instead of making copies of it.

One problem: with the tools we've seen so far, there is no good way to tell when it's possible to move/steal data instead of copying it.

We can move out of create_an_x() but not out of x2. **Why?**

LValues and RValues allow us to distinguish between temporary and named data

Rvalues are values that can only live on the right hand side of an assignment operator--they have no named location in memory.

C++ lets us overload functions on the value category of the input with **rvalue references**, which can only bind to rvalues

```
ETC(ETC&& other) noexcept : data{nullptr}, size{0} {
  std::swap(data, other.data);
  std::swap(size, other.size);
}
```

Surprising Side Effect: Replacing a variable with an expression of its value can now sometimes fail!

```
int main(){
  int x = 5;
  do_something(5); // Works!
  do_something(x); // Compiler Error!
}
```

RValue references are almost exclusively used to implement move semantics

Since an rvalue can't be referred to again, we can just steal all of its data!

This is called **move semantics** and is implemented by making a **move constructor** and **move assignment operator.**

```
ETC(ETC&& other) noexcept : data{nullptr}, size{0} {
  std::swap(data, other.data);
  std::swap(size, other.size);
}
```

Quiz Next Week!

Vote on Piazza if you want it to be on Canvas or on paper

Focus is mostly on iterators/STL, with a lesser focus on templates

See the last slides in this presentation for a list of what to study

Project 3

Infinite lazy streams

Have you ever wanted to build a list of all the prime numbers?

Well now you can!

Project 3

Infinite lazy streams

The most challenging project to date! Requires knowledge of:

- Templates
- Shared Pointers (next lecture)
- Rvalue/Lvalue references
- Perfect forwarding (next lecture)
- Classes/Objects/Inheritance

And even then, strange bugs will pop up (e.g. segfaults due to accidental infinite recursion)

Depending on your background, 1.5x to 4x harder than Project 2

Notecards

- Name and EID
- One thing you learned today (can be "nothing")
- One question you have about the material. **<u>If you leave this blank, you will be docked points.</u>**

  If you do not want your question to be put on Piazza, please write the letters **NPZ** and circle them.

Quiz 3

You should know:

- What templates are
- What parametric polymorphism is and how it differs from ad-hoc polymorphism
- The basics of template syntax
- When template code is actually generated
- Code layout rules when using templates
- Why iterators are needed
- The interface of an iterator (i.e. what each member does/is)
- The iterator capability hierarchy
- The special iterators insert and reverse, and what they do
- The names and parts of a C++ lambda
- How captured variables are treated in a lambda
- When it is legal to use variables in a lambda

You ***do not*** need to (know):

- Mechanisms of template code generation
- decltype/declval
- How to use templates with anything but typename in the template argument (i.e. template metaprograms)

You should be able to:

- Write a simple template function
- Understand how to implement a simple iterator for a data structure
- Read the function signature for a function in <algorithm> and be able to describe what it does.

Additional Resources

- A short guide on rvalue references, move semantics, and forwarding
- An extended guide on rvalue references, their motivation, and the forwarding problem
- Eli Bendersky's guide on rvalues and lvalues (more detail on rvalues/lvalues, and not so much about move/forward)
- Another great short guide on rvalue references and move semantics
- Yet another short guide on move semantics (this one linked from the ISO CPP guide!)
- Stack Overflow question on move semantics
- Universal references and how they differ from rvalue references