Some features in C++ 11

rvalue

- These are temporary anonymous objects whose lifetime is the statement in which they are created.
  - They don't have a name (hence anonymous)
  - Lifetime is the statement where they are created(hence temporary)
- Example:
  - vector <int> GetValues();
    - » The return value of this function is a vector by value.
    - » This could be a temporary value created for the return.

rvalue

- Consider the assignment operator below:

```
MyClass & operator = ( MyClass other )          // pass by value
 {
        swap( other );
}
```

We swap *this* with the temporary passed by value copy (since we don't care about the temp copy 'other')

- Now, what if we passed in a temporary to this operator:

```
MyClass Foo();      // Foo is a function that returns MyClass object

MyClass m1;
m1 = Foo();         // Return type of Foo is an unnamed temporary.
                    // Temporary passed to assign operator,
                    // and a copy is made (pass by value)
```

rvalue

Now consider this different assignment operator:

```
MyClass & operator = ( MyClass && other )        // rvalue reference
{
        swap( other );
}
```

If we have this move assignment operator, then we don't have to create a copy for passing, and we can still do the move (calling swap inside).

rvalue

So,

X && is an rvalue reference

X & is an lvalue reference.

- Think of lvalue as something that you can get an address for using the & operator.
- And in case of rvalue, you cannot do that.

- These help with function overload resolution.
- A function Foo taking an rvalue reference will be chosen when it is passed an rvalue.
- Its overloaded Foo taking an lvalue reference will be chosen when it is passed an lvalue.

rvalue

```
void ProcessStr( string && str )                    // takes rvalue reference
{}

void ProcessStr( string & str )                     // takes lvalue reference
{}

void Foo()
{
        ProcessStr( "ABC" );
        ProcessStr( string( "123" ));

        string myStr("XYZ");
        ProcessStr(  myStr );
}
```

move constructor

- Takes rvalue reference.
- Will get called when you are passing an anonymous temporary.
- Will typically move (shallow copy) resources from the passed in rvalue reference.

vector push_back

- We had talked about the vector class earlier, and also about strong exception guarantees.
- vector's push_back member function provides a strong exception guarantee.
  - It can do that because it is only appending at the end.
  - And in case it has to reallocate to grow,
    - it can allocate new memory,
    - copy the objects over,
    - If no exception so far, then delete old memory at the end and make vector internals to point to new memory.

vector push_back

- But with move semantics, you have to keep some things in mind.

- If push_back reallocates and now "moves" the existing objects to the new memory, then it is possible that the move throws an exception.
- And if the exception happens, its original memory block has changed, since it moved the objects from original memory to new memory.
- And so the strong exception guarantee cannot be offered in this case.

- Now, if the move constructor is declared as noexcept (i.e., it does not throw), then the strong guarantee can be given.
- push_back calls move_if_noexcept.

- So, strong guarantee is given if the objects are no throw move constructible, or copy constructible.

std:: move_if_noexcept

- std::move_if_noexcept
  - Cast to rvalue if:
    - Move constructor does not throw (is declared as noexcept)
    - Or the type is not copy constructible.
  - Else
    - return as lvalue

noexcept

- This is how you specify that a function does not throw an exception.
- Specifying a function as noexcept can enable some compiler optimizations.
- template <typename T> void Foo (const char *) noexcept (true);
  - Same as: template <typename T> void Foo (const char *) noexcept;
- template <typename T> void Foo (const char *) noexcept (false);
  - Same as : template <typename T> void Foo (const char *)

noexcept

- noexcept can take an expression as well:

```
template <typename T> void Foo( T & t1 ) noexcept ( noexcept ( ++ t1)  );

template <typename T> void Foo ( T & t1 ) noexcept ( noexcept (*t1)  );

struct pair
{
   void swap( pair & other ) noexcept ( noexcept(swap(first, other.first)
                                          && swap(second, other.second ) ) );
};
```