Programming in Modern C++

Module M49: C++11 and beyond: General Features: Part 4: Rvalue and Move/1

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Introduced following **C++11** general features:
  - constexpr (+ **C++14**)
  - noexcept
  - nullptr
  - Inline namespace
  - `static_assert`
  - User-defined Literals (+ **C++14**)
  - Digit Separators and Binary Literals (+ **C++14**)
  - Raw String Literals
  - Unicode Support
  - Memory Alignment
  - Attributes (+ **C++14**)

Module Objectives

- Understanding the difference between Copying and Moving
- Understanding the difference between Lvalue and Rvalue
- Exploiting the advantages of Move in C++ using
  - Rvalue Reference
  - Move Semantics
  - Copy / Move Constructor / Assignment
  - Implementation of Move Semantics

Module Outline

Copying vs. Moving

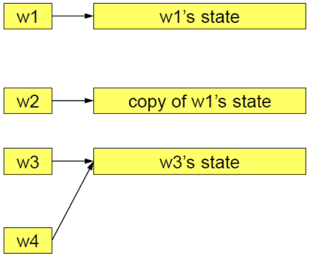
Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

Copying vs. Moving

- C++ has always supported copying object state:
  - *Copy* constructors, *Copy* assignment operators
- C++11 adds support for requests to *Move* object state:

```
Widget w1;
...
// copy w1's state to w2
Widget w2(w1);
Widget w3;
...
// move w3's state to w4
Widget w4(std::move(w3));
```

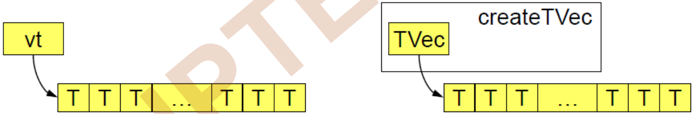


- **Note**: w3 continues to exist in a valid state after creation of w4

Copying vs. Moving: Return Value

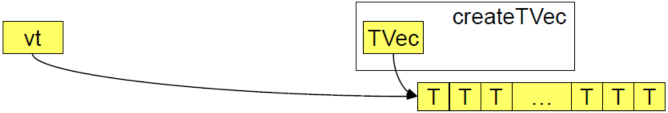
- C++ at times performs *extra copy*, while *temporary objects* are prime candidates for *move*:

```
typedef std::vector<T> TVec;
TVec createTVec(); // factory function
TVec vt;
...
vt = createTVec(); // in C++03, copy return value to vt, then destroy return value
```



- *Moving* values would be cheaper and C++11 generally turns such copy operations into moves:

```
TVec vt;
...
vt = createTVec(); // implicit move request in C++11. move data in return value object to vt
                   // then destroy return value object
```

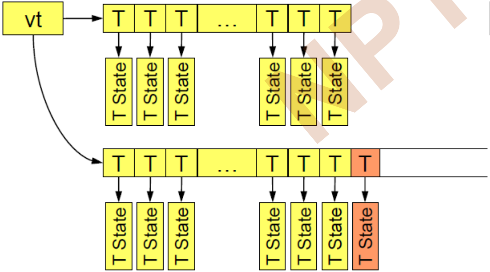


Copying vs. Moving: Append a Full Vector

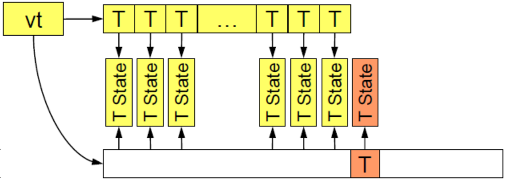
- Appending to a full vector causes much *copying before the append*. Moving would be efficient:

assume vt lacks unused capacity

```
std::vector<T> vt;
...
vt.push_back(T object);
```




```
std::vector<T> vt;
...
vt.push_back(T object);
```



- vector and deque operations like insert, emplace, resize, erase, etc. would benefit too

Copying vs. Moving: Swap

- Consider swapping two values:

By Copy

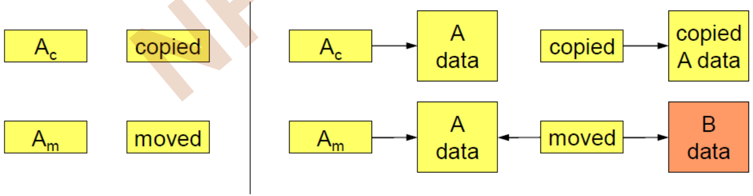
```
template<typename T> void swap(T& a, T& b) { // std::swap impl. by copy
void swap(T& a, T& b) {
    T tmp(a);           // copy a to tmp (=> 2 copies of a)
    a = b;              // copy b to a (=> 2 copies of b)
    b = tmp;            // copy tmp to b (=> 2 copies of tmp)
}                       // destroy tmp
```

By Move

```
template<typename T> void swap(T& a, T& b) { // std::swap impl. by move
    T tmp(std::move(a)); // move a's data to tmp
    a = std::move(b);    // move b's data to a
    b = std::move(tmp);  // move tmp's data to b
}                        // destroy (eviscerated) tmp
```

Copying vs. Moving: Deep vs. Shallow Copy

- Moving most important when:
  - *Object has data in separate memory (for example, on free store).*
  - *Copying is deep*
- Moving copies only object memory
  - Copying copies object memory + **separate memory**
- Consider copying/moving A to B:



- **Moving never slower than copying, and often faster**

Simple Performance Test

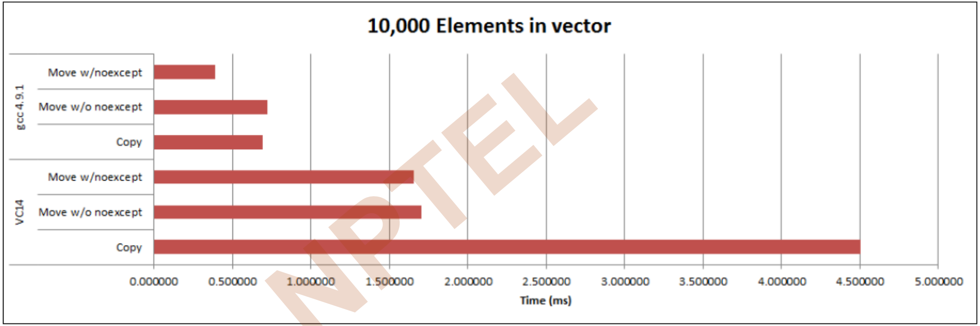
- Given

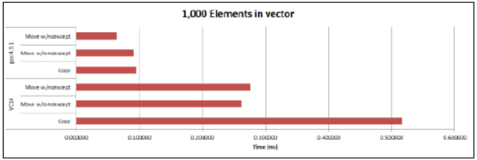
```
const std::string stringValue("This string has 29 characters");

class Widget { std::string s;
public:
    Widget(): s(stringValue) { }
    ...
};
```

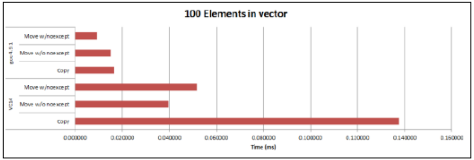
- Consider this push_back-based loop:

```
std::vector<Widget> vw;
Widget w;
for (std::size_t i = 0; i < n; ++i) {  // append n copies of w to vw
    vw.push_back(w);
}
```

Performance Data







Copying vs. Moving

- Lets C++ recognize move opportunities and take advantage of them.
  - **How recognize them?**
  - **How take advantage of them?**
- **Moving a key new C++11 idea**
  - Usually an optimization of copying
- Most standard types in C++11 are *move-enabled*
  - They support move requests
  - For example, STL containers
- Some types are *move-only*:
  - Copying prohibited, but moving is allowed
  - For example, stream objects, `std::thread objects`, `std::unique_ptr`, etc.

Rvalue References and Move Semantics

Sources:

- Rvalue references and move semantics, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Rvalue References
  - C++ Rvalue References Explained
  - Lvalues and Rvalues, accu.org, 2004
  - What are rvalues, lvalues, xvalues, glvalues, and prvalues?, stackoverflow.com, 2010
- Move Semantics
  - What is move semantics?, stackoverflow.com, 2010
  - M.3 — Move constructors and move assignment, learncpp.com, 2021
  - Move Constructors and Move Assignment Operators (C++), Microsoft, 2021
  - Understanding Move Semantics and Perfect Forwarding: Part 1, Part 2: Rvalue References and Move Semantics, Part 3: Perfect Forwarding, Drew Campbell, 2018

Rvalue References and Move Semantics

Lvalues and Rvalues

- **Lvalues** *are generally things we can take the address of:*
  - In C, Expressions on *left-hand-side (LHS)* of an assignment
  - Named objects - variables
  - Legal to apply address of (&) operator
  - Lvalue references
- **Rvalues** *are generally things we cannot take the address of:*
  - In C, Expressions on *right-hand-side (RHS)* of an assignment
  - Typically unnamed temporary objects - expressions, return values from functions, etc.
  - Rvalue references
- Examples:

```
int x, *pInt;                     // x, pInt, *pInt are lvalues
std::size_t f(std::string str); // f and str are lvalues, f's return is rvalue
f("Hello");                       // temp string("Hello") created for call is rvalue
std::vector<int> vi;              // vi is lvalue
...
vi[5] = 0;                        // vi[5] is lvalue
```

  - Recall that `vector<T>::operator[]` returns `T&`

Moving and Lvalues

- *Value movement generally not safe when the source is an lvalue object*
  - That continues to exist, may be referred to later:

```
TVec vt1;
...
TVec vt2(vt1);      // it is expected that vt1 be copied to vt2, not moved!
...use vt1...       // value of vt1 here should be same as above
```

- *Value movement is safe when the source is an rvalue object*
  - Temp's usually *go away at statement's end*. No way to tell if their *value has been modified*

```
TVec createTVec();                // as before
TVec vt1;
vt1 = createTVec();               // rvalue source: move okay
auto vt2 = createTVec();          // rvalue source: move okay
vt1 = vt2;                        // lvalue source: copy needed
auto vt3(vt2);                    // lvalue source: copy needed
std::size_t f(std::string str);   // as before
f("Hello");                       // rvalue (temp) source: move okay
std::string s("C++11");
f(s);                             // lvalue source: copy needed
```

Rvalue References

- C++11 introduces **rvalue references**
  - Syntax: `T&&`
  - **Normal references** now known as **lvalue references**
    - ▷ Rvalue references behave similarly to Lvalue references
  - Must be initialized, cannot be rebound, etc.
- **Rvalue references identify objects that may be moved from**
- Reference Binding Rules
  - Important for overloading resolution
  - As always:
    - ▷ Lvalues may bind to *lvalue references*
    - ▷ Rvalues may bind to *lvalue references to* `const`
  - In addition:
    - ▷ Rvalues may bind to *rvalue references to non-*`const`
    - ▷ Lvalues may *not* bind to *rvalue references*
      - – Otherwise lvalues could be accidentally modified

Rvalue References

- Examples:

```
void f1(const TVec&);        // takes const lvalue ref
TVec vt;
f1(vt);                      // fine (as always)
f1(createTVec());            // fine (as always)
void f2(const TVec&);        // #1: takes const lvalue ref
void f2(TVec&&);             // #2: takes non-const rvalue ref
f2(vt);                      // lvalue => #1
f2(createTVec());            // both viable, non-const rvalue => #2
void f3(const TVec&&);       // #1: takes const rvalue ref
void f3(TVec&&);             // #2: takes non-const rvalue ref
f3(vt);                      // error! lvalue
f3(createTVec());            // both viable, non-const rvalue => #2
```

Rvalue References and const

- C++ remains `const`-correct:
  - `const` lvalues / rvalues bind only to references-to-`const`
- But rvalue-references-to-`const` are essentially useless
  - Rvalue references designed for two specific problems:
    - **Move semantics**
    - **Perfect forwarding**
  - C++11 language rules carefully crafted for these needs
    - Rvalue-refs-to-`const` not considered in these rules
  - `const T&&`s are legal, but not designed to be useful
    - Uses already emerging :-)
- Implications:
  - Do not declare `const T&&` parameters
  - Not possible to move from them, anyway
  - Hence this rarely makes sense:

Distinguishing Copying from Moving

- Overloading exposes move-instead-of-copy opportunities:

```
class Widget { public:
    Widget(const Widget&);                    // copy constructor
    Widget(Widget&&) noexcept;                // move constructor
    Widget& operator=(const Widget&);         // copy assignment op
    Widget& operator=(Widget&&) noexcept;     // move assignment op
  ...
};
Widget createWidget(); // factory function
Widget w1;
Widget w2 = w1;        // lvalue src => copy required
w2 = createWidget();   // rvalue src => move okay
w1 = w2;               // lvalue src => copy required
```

- Move operations need not be noexcept, but it is preferable
  - Moves should be fast, and noexcept => more optimizable
  - Some contexts require noexcept moves (for example, std::vector::push_back)
  - Move operations often have natural noexcept implementations
- We declare move operations noexcept by default

Copy vs. Move : Lvalue vs. Rvalue

```
class A { public: A() { std::cout << "Defa Ctor" << endl; }                     // Defa Constructor
    A(const A&) { std::cout << "Copy Ctor" << endl; }                            // Copy Constructor
    A(A&&) noexcept { std::cout << "Move Ctor" << endl; }                        // Move Constructor
    A& operator=(const A&) { cout << "Copy =" << endl; return *this; }           // Copy =
    A& operator=(A&&) noexcept { cout << "Move =" << endl; return *this; }       // Move =
    friend A operator+(const A& a, const A& b) { A t; return t; } // Temp. obj. ret.-by-value
};
```

	Only Copy: Debug	Only Copy: Release	Copy & Move: Debug	Copy & Move: Release
`A a;                 // lvalue`	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
`A b = a;             // lvalue`	Copy Ctor	Copy Ctor	Copy Ctor	Copy Ctor
`A c = a + b;         // rvalue`	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
`// RVO in a + b for release build`	Copy Ctor	// RVO	Move Ctor	// RVO
`A d = std::move(a);  // rvalue`	Copy Ctor	Copy Ctor	Move Ctor	Move Ctor
`b = a;               // lvalue`	Copy =	Copy =	Copy =	Copy =
`c = a + b;           // rvalue`	Defa Ctor	Defa Ctor	Defa Ctor	Defa Ctor
	Copy Ctor	// RVO	Move Ctor	// RVO
	Copy =	Copy =	Move =	Move =

- Return Value Optimization (RVO) eliminates the temp. obj. created to hold a function's return value
- `std::move(t)` produces a rvalue from `t` to indicate that the object `t` may be *moved from*

Copy vs. Move : Lvalue vs. Rvalue: Explanation

```
class A { public: A() { std::cout << "Defa Ctor" << endl; }              // Defa Constructor
    A(const A&) { std::cout << "Copy Ctor" << endl; }                   // Copy Constructor
    A(A&&) noexcept { std::cout << "Move Ctor" << endl; }               // Move Constructor
    A& operator=(const A&) { cout << "Copy =" << endl; return *this; }     // Copy =
    A& operator=(A&&) noexcept { cout << "Move =" << endl; return *this; } // Move =
    friend A operator+(const A& a, const A& b) { A t; return t; } // Temp. obj. ret.-by-value
};
```

- `A a;` ⇒ `a` is an lvalue and is *default constructed*
- `A b = a;` ⇒ `a` is an lvalue and hence, `b` is *copy constructed*
- `A c = a + b;` ⇒ `operator+(a, b)` *default constructs* `t`, computes the result of `a + b` in `t` (not shown) and then *returns* `t` *by value*. Hence `a + b` is an rvalue and `c` is *move constructed* (if available, else *copy constructed*). **Note that in release (optimized) compiler build, RVO[1] allows `t` to be constructed directly in `c` and no copy or move construction is needed**
- `A d = std::move(a);` ⇒ `std::move` (in `<utility>`) can force an rvalue type. It produces an rvalue from `t` to indicate that the object `t` may be *moved from*. Hence `d` is *move constructed* (if available, else *copy constructed*)
- `b = a;` ⇒ `a` is an lvalue and hence, `b` is *copy assigned*
- `c = a + b;` ⇒ As above, `c` is *move assigned* (if available, else *copy assigned*) after *move construction* (if available, else *copy construction*). *Copy (move) construction* is eliminated by RVO in release build

[1] Return Value Optimization (RVO) eliminates the temp. obj. created to hold a function's return value

Copy vs. Move : Vector


```
// C++ program with the copy and the move constructors

class C { int* data; // Declare the raw pointer as the data member of class
public:
    C(int d) {                // Constructor
        data = new int(d); // Declare object in the heap
        cout << "Ctor: " << d << endl;
    };
    C(const C& src) : myClass{ *src.data } { // Copy Constructor by delegation
        // Copying the data by making deep copy
        cout << "C-Ctor: " << *src.data << endl;
    }
    C(C&& src) : data{ src.data } noexcept { // Move Constructor
        cout << "M-Ctor: " << *src.data << endl;
        src.data = nullptr;
    }
    ~C() { // Destructor
        if (data != nullptr) // If pointer is not pointing to nullptr
            cout << "Dtor: " << *data << endl;
        else                 // If pointer is pointing to nullptr
            cout << "Dtor: " << "nullptr " << endl;
        delete data; // Free up the memory assigned to the data member of the object
    }
};
```

Copy vs. Move : Vector

```
int main() { vector<C> v; // Create vector of C Class
    v.push_back(C{10});   // Inserting object of C class
    v.push_back(C{20});
}
```

	Only Copy			Copy & Move		
	Debug	*Release*	*Remark*	*Debug*	*Release*	*Remark*
`{ vector<C> v;`						

`// v.size() = 0`
`v.push_back(C{10});`
`// v.size() = 1`
	Ctor: 10
	Ctor: 10
	C-Ctor: 10
	Dtor: 10
		Ctor: 10
		Ctor: 10
		C-Ctor: 10
		Dtor: 10
			
			// Delegate
			// C-Ctor
				Ctor: 10
				
				M-Ctor: 10
				Dtor: nullptr
					Ctor: 10
					
					M-Ctor: 10
					Dtor: nullptr
						
						// Add 10 to v

`// Move C{10}`
`// for C{20}`
`v.push_back(C{20});`
`// v.size() = 2`
	Ctor: 20
	Ctor: 10
	C-Ctor: 10
	Dtor: 10
	Ctor: 20
	C-Ctor: 20
	Dtor: 20
		Ctor: 20
		Ctor: 10
		C-Ctor: 10
		Dtor: 10
		Ctor: 20
		C-Ctor: 20
		Dtor: 20
			
			// Delegate
			// C-Ctor
			
			// Delegate
			// C-Ctor
				Ctor: 20
				
				M-Ctor: 10
				Dtor: nullptr
				
				M-Ctor: 20
				Dtor: nullptr
					Ctor: 20
					
					M-Ctor: 10
					Dtor: nullptr
					
					M-Ctor: 20
					Dtor: nullptr
						
						// Move 10 in v
						
						// Add 20 to v

`// End of scope` `} // Release v`	Dtor: 10 Dtor: 20	Dtor: 10 Dtor: 20	// Release // Vector v	Dtor: 10 Dtor: 20	Dtor: 10 Dtor: 20	// Release // Vector v

Copy vs. Move : Vector: Explanation

```
class C { int* data; /* raw pointer */ public:
C(int d); /*Ctor*/ C(const C& src); /*C-Ctor*/ C(C&& src); /*M-Ctor*/ ~C(); /*Dtor*/ };
```

- `{ vector<C> v;` ⇒ `v` is *default constructed* as an empty vector of `C`. `v.size() = 0`
- `v.push_back(C{10});` ⇒ Construct `C{10}`, copy/move & place in `v[0]`, and destruct. `v.size() = 1`

```
Ctor: 10    /* Ctor for C{10} => t10, Temp.obj. and rvalue             */ Ctor: 10
Ctor: 10    /* delegated from C-Ctor                                   */
C-Ctor: 10  /* C-Ctor for t10 => v10 = v[0], lvalue to place in v      */ M-Ctor: 10
Dtor: 10    /* Dtor for t10                                            */ Dtor: nullptr
```

- `v.push_back(C{20});` ⇒ Construct `C{20}`. Copy/move `v[0]` and destruct old `v[0]`. Copy/move & place `C{20}` in `v[1]`, and destruct. `v.size() = 2`

```
Ctor: 20    /* Ctor for C{20} => t20, Temp.obj. & rvalue               */ Ctor: 20
Ctor: 10    /* delegated from C-Ctor                                   */
C-Ctor: 10  /* C-Ctor for v10 => v10_1 = v[0], lvalue to place in v    */ M-Ctor: 10
Dtor: 10    /* Dtor for v10                                            */ Dtor: nullptr
Ctor: 20    /* delegated from C-Ctor                                   */
C-Ctor: 20  /* C-Ctor for t20 => v20 = v[1], lvalue to place in v      */ M-Ctor: 20
Dtor: 20    /* Dtor for t20                                            */ Dtor: nullptr
```

- `}` ⇒ Automatic `v` going out of scope. Destruct `v[0]` and `v[1]`

```
Dtor: 10    /* Dtor for v10_1 = v[0]                                   */ Dtor: 10
Dtor: 20    /* Dtor for v20 = v[1]                                     */ Dtor: 20
```

Copy vs. Move : Vector: Performance Trade-off

- Since, class `C` has no default constructor, `vector<C> v` is constructed as an empty vector with `v.size() = 0`. Hence, every time a `push_back` (`insert` at the `end()`) is done, we need to expand the allocation of the vector by copying / moving the existing elements
- For `v.push_back(C{10})`, `C{10}` is constructed as a temporary object (rvalue). So, it needs to be copied / moved for `push_back` to the vector as lvalue. Same for `v.push_back(C{20})`
- Further, for `v.push_back(C{20})`, fresh allocation and copy / movement of existing element is needed for `push_back`
- To `push_back` the n^{th} element, we need to copy / move existing $n-1$ elements. This means:
  - **Using Copy**
    - $n-1$ resource allocations (new int) and de-allocations (delete)
    - For n elements this adds to $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$ total allocations / de-allocations
  - **Using Move**
    - 0 resource allocations (new int) and de-allocations (delete)
    - For n elements this adds to $\sum_{i=0}^{n-1} 0 = 0$ total allocations / de-allocations. **Huge Benefit!**

Implementing Move Semantics

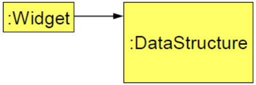
Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

Implementing Move Semantics

- Move operations take source's value, but leave source in valid state:

```
class Widget {
public:
    Widget(Widget&& rhs) noexcept : pds(rhs.pds) // take source's value
        { rhs.pds = nullptr; }                   // leave source in valid state

    Widget& operator=(Widget&& rhs) noexcept {
        delete pds;          // get rid of current value
        pds = rhs.pds;       // take source's value
        rhs.pds = nullptr;   // leave source in valid state
        return *this;
    }
    ...

private:
    struct DataStructure;
    DataStructure *pds;
};
```



- Easy for built-in types (for example, pointers). Trickier for UDTs. . .

Implementing Move Semantics

- Widget's move operator= fails given move-to-self:

```
Widget w;
w = std::move(w); // undefined behavior!
```

- It may be harder to recognize, of course:

```
Widget *pw1, *pw2;
...
*pw1 = std::move(*pw2); // undefined if pw1 == pw2
```

- C++11 condones this
    - In contrast to copy operator=
- A fix is simple, if you are inclined to implement it:

```
Widget& Widget::operator=(Widget&& rhs) noexcept {
    if (this == &rhs) return *this; // or assert(this != &rhs);
    ...
}
```

Implementing Move Semantics

- Part of C++11's `string` type:

```
string::string(const string&);       // copy constructor
string::string(string&&) noexcept;   // move constructor
```

- An incorrect move constructor:

```
class Widget { std::string s;
public:
    Widget(Widget&& rhs) noexcept // move constructor
        : s(rhs.s) // compiles, but copies!
    { ... }
    ...
};
```

- `rhs.s` an lvalue, because it has a name
  - Lvalueness / Rvalueness orthogonal to type!
    - `int`s can be lvalues or rvalues, and rvalue references can, too.
  - `s` initialized by `string`'s *copy* constructor

Implementing Move Semantics

- Another example:

```
class WidgetBase { public:
    WidgetBase(const WidgetBase&);      // copy ctor
    WidgetBase(WidgetBase&&) noexcept; // move ctor
    ...
};

class Widget: public WidgetBase { public:
    Widget(Widget&& rhs) noexcept      // move ctor
    : WidgetBase(rhs)                  // copies!
    { ... }
    ...
};
```

- rhs is an lvalue, because it has a name
  - Its declaration as Widget&& not relevant!

Module Summary

- Understood the difference between Copying and Moving
- Understood the difference between Lvalue and Rvalue
- Learnt the advantages of Move in C++ using
  - Rvalue Reference
  - Move Semantics
  - Copy / Move Constructor / Assignment
  - Implementation of Move Semantics