Introduction

This document provides a basic introduction to C++ copy and move semantics. It should be used in conjunction with the example code file CopyAndMoveSemanticsBasics.cpp

For any given topic discussed in regards to copy and move semantics, there are additional important details not discussed because this is a basic introduction. There are numerous official online resources for C++ that can be consulted for these details. Links to some of these resources, specifically *Microsoft*, *C++ Reference*, and the I*SO C++ Standard*, are provided at the end in the **Resources** section.

- Microsoft and C++ Reference provide a general reference for any C++ related topic. Microsoft's website is a bit easier to understand and more beginner friendly.
- The ISO C++ Standard contains the C++ Core Guidelines which is a discussion on the current C++ best practices and standards. It is not meant for learning about a C++ topic but rather meant for someone who already knows about a specific C++ topic, and is looking for a further discussion on best practices.

Note: For the purposes of demonstration, the std namespace is being used by including the statement using namespace std. This is just to make code easier to read. For example, it allows string to be explicitly written instead of std::string to refer to a C++ string, and it allows *cout* to be explicitly written instead of std::*cout*. It is okay to use the std namespace for the purposes of learning, teaching, or demonstration, but in real C++ code it is not a good practice.

Table of Contents

Topics	**Page**
Lvalues and Rvalues	2
Copy Semantics	4
Move Semantics	6
The Rule of Three/Five/Zero	8
Resources	9

Lvalues and Rvalues

Introduction to lvalues and rvalues

Though **lvalues** and **rvalues** have official definitions, the best way to think of them in general is that an lvalue is something that you can reference by a name, and an rvalue is something that you cannot reference by a name.

- **lvalues** take the form of things like variables and objects.
- **rvalues** take the form of things like hard coded numbers and temporary objects

In the statement

```
int x = 3;
```

x is an lvalue since it's a variable with a name, whereas 3 is an rvalue since it's just the number 3. It is not a variable but rather a temporary value loaded somewhere in the computer's memory for the purposes of executing that one line of code.

An lvalue can go on the left and right side of the assignment operator, and it has utility in both places. An rvalue can go on the right hand side of the assignment operator and it has utility there. Some rvalues can go on the left hand side of the assignment operator and some can't. Regardless, for the ones that can go on the left hand side of the assignment operator, they have zero utility. The following are valid examples.

`int x = 3;`	x is an *lvalue*, 3 is an *rvalue*
`int y = x;`	x and y are both *lvalues*
`int *px = &x;`	px is an *lvalue*, &x (address of x) is an *rvalue*
`int py = &y;`	py is an *lvalue*, &y (address of y) is an *rvalue*
`py = px;`	py and px are both *lvalues*
`string s = "Liebestraum";`	s is an *lvalue*, "Liebestraum" is an *rvalue*
`string s1 = string("Liebestraum");`	s1 is an *lvalue*, string("Liebestraum") is an *rvalue*
`string s2 = s;`	s2 and s are both *lvalues*

The following are invalid examples since there are rvalues on the left hand side of the assignment operator.

`3 = x;`	invalid
`&x = px;`	invalid
`"Liebestraum" = s;`	invalid
`x * y = 3;`	invalid

As seen in the above examples, rvalues typically can't go on the left hand side of the assignment operator. However, there are unique cases where they can, such as in the example below. Note that examples such as the one below have zero utility and there's no point in doing them. However, it is still valid C++ code that will compile.

`s + s = s;`	s + s is an *rvalue*, s is an *lvalue*

lvalue and rvalue reference declarators

The examples on the previous page give a good overview of lvalues and rvalues, but the picture is still incomplete. The final component to understanding lvalues and rvalues (and understanding copy and move semantics) is introducing the **lvalue reference declarator** denoted by & and the **rvalue reference declarator** denoted by &&.

The **lvalue reference declarator** means something is a reference to an already existing variable or object. For example:

```
int x = 3;
int& rx = x;
```

In the above example, rx is a reference to an already existing integer named x. Therefore, by modifying rx, x actually gets modified. Similarly, by printing rx, x is actually getting printed.

```
x = 7;          // x is now 7
rx = 9;         // x is now 9
cout << x;      // prints 9, the value stored in x
cout << rx;     // prints 9, the value stored in x
```

lvalue reference declarators can exist for objects as well.

```
string s = "Liebestraum";
string& rs = s;
```

Again, in the above example, rs is a reference to an already existing string named s. Once again, by modifying rs, s actually gets modified. Similarly, by printing rs, s actually gets printed

```
s = "Liszt";            // s is now Liszt
rs = "Liebestraum";     // s is now Liebestraum again
cout << s;              // prints "Liebestraum", the string stored in s
cout << rs;
```

To avoid confusion, it should be noted that & has been seen in two different contexts in this document with two different meanings up until this point. The & operator can be used as the address operator and as an lvalue reference. It's used as the address operator when applied to an already existing variable/object, and it's used as the lvalue reference when declaring a new variable/object

```
int x = 3;
int* px = &x;           // address operator
int& rx = x;            // lvalue reference
```

In the above example, px is a pointer to x (so it holds the address of x), whereas rx is a reference to x so it just refers to the variable x. Putting it all together, x can be modified in 3 ways. All of the lines below would store the value of 7 in x.

```
x = 7;
*px = 7;
rx = 7;
```

The **rvalue reference declarator** means something is an rvalue reference. It's utility is the context of move semantics which is discussed later in this document. Putting them both together, lvalue references are used for copy semantics, and rvalue references are used for move semantics.

Copy Semantics

Copy semantics are a reference to the **copy constructor** and the **copy assignment operator** (the overloaded = operator) for a class. Both the copy constructor and copy assignment operator create a complete and independent copy of one object and store it in another object. This is also called a **deep copy**. The difference is that the copy constructor is a constructor and therefore is used only when an object is first instantiated. The copy assignment operator is used on an already existing object.

Copy Constructor

For some class named MyClass with the following member variables

```
private:
    int x;
    int* a;
```

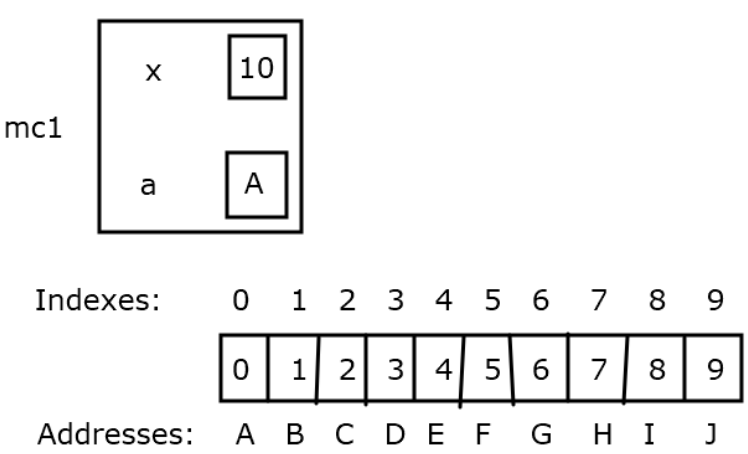
the format of the copy constructor is as follows

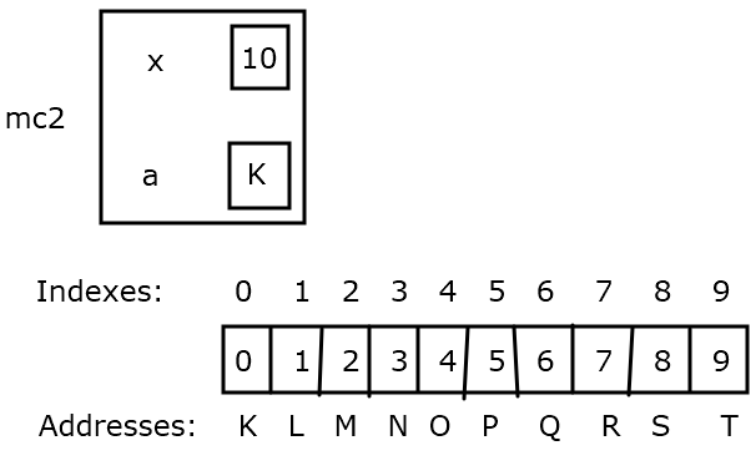
```
MyClass(const MyClass& myClass);
```

This means that a complete and independent copy of myClass is going to be created and stored in the new object being instantiated. The & means that myClass is an lvalue reference - it is a reference to an existing object. The const means that the internal representation of myClass will not be changed during the constructor call. Again, myClass is being copied, but nothing within myClass is actually changing. An example of a call to the copy constructor is below:

```
MyClass mc1;            // default constructor
MyClass mc2(mc1);       // copy constructor
MyClass mc3 = mc1;      // copy constructor
```

An object named mc1 is created with the default constructor. Then, mc2 is created with the copy constructor. Below, it can be seen how mc2 is a is a deep copy of mc1.





They don't share the same array - mc1 has one array, and mc2 has a completely separate array that is an exact copy of mc1's array. The x value was also copied. The second example is a little more confusing. It includes "mc3 = mc1", so it would be assumed that this was in fact calling the copy assignment operator. However, because the = operator is being used when the object is being instantiated, what actually gets called is the copy constructor.

Copy Assignment Operator

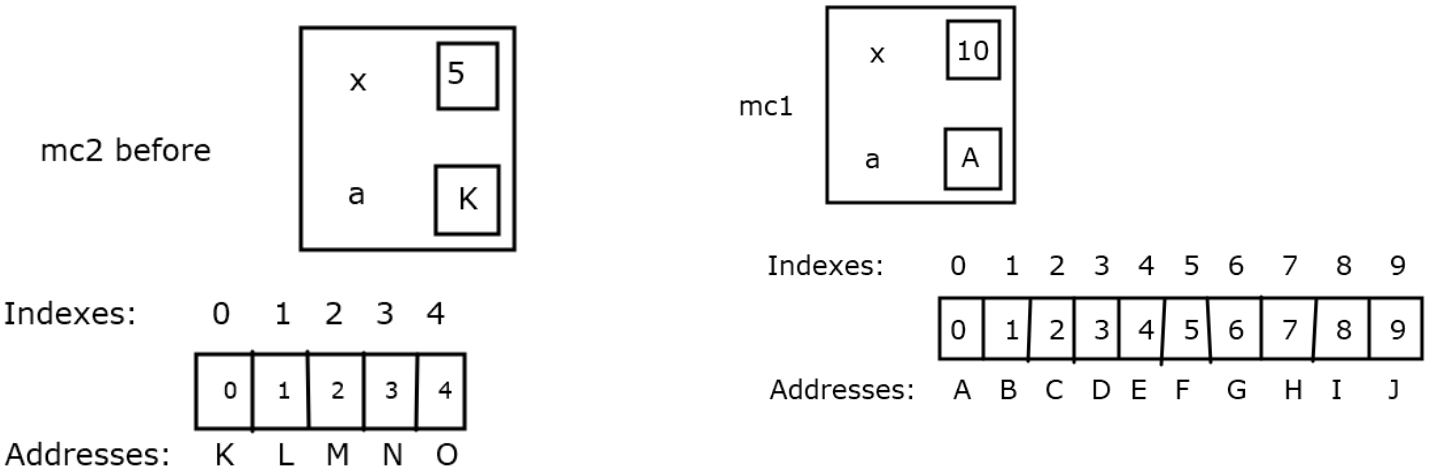
For MyClass, the format of the copy assignment operator is

```
MyClass& operator=(const MyClass& myClass);
```

Again, This means that a deep copy of myClass is going to be created and stored in an already existing object, and the & means that myClass is an lvalue reference - it is a reference to an existing object. The const means that the internal representation of myClass will not be changed during the constructor call.

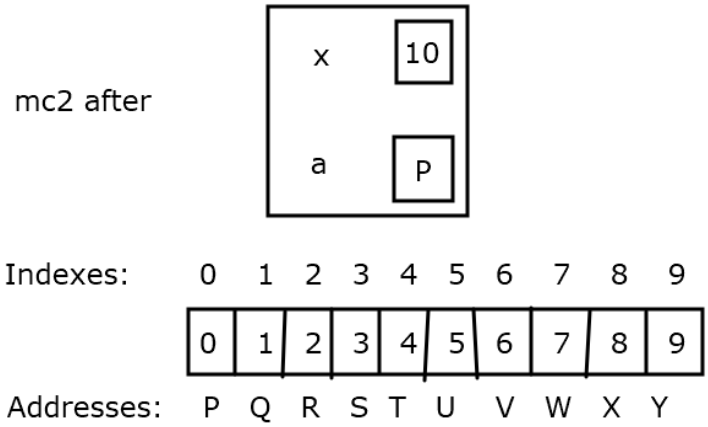
```
MyClass mc1;
MyClass mc2(5);
mc2 = mc1;                    // copy assignment operator on an existing object
```

Before Copy Assignment Operator



After Copy Assignment Operator

As with the copy constructor, mc2 is now a deep copy of mc1. Since mc2 already existed, it's array before (starting with address K) had to be deallocated before creating an array that is a copy of mc1's array. If this were not done, it would have caused a memory leak.



Move Semantics

Move semantics are a reference to the **move constructor** and the **move assignment operator** The move assignment operator is a different version of the overloaded assignment operator = for a class. Instead of making a copy of one object and storing it another object, the move constructor and move assignment operator transfer the content of one object and store it in another - there is no copying being done, it is simply the transfer of the internal representation of one object to another. Move semantics exist because it prevents unnecessary copy operations. If object *b* needs a copy of the data in object *a*, and afterwards *a* doesn't need the data anymore, then to create a whole new copy of *a* would be a waste of time, especially if a held a lot data. Instead, just transfer the data from *a* to *b*.

Note that the object that had its data transferred from must be left in a valid default state. Any pointers should be set to nullptr, any objects (like string) can be set to their empty state, and it would make sense to set primitive types (like int or double) to a default value of zero.

Move semantics required that the object whose data is being transferred be an rvalue reference. For the case where the object is an lvalue, the std::*move( )* function must be used to first turn the lvalue into an rvalue.

Move Constructor

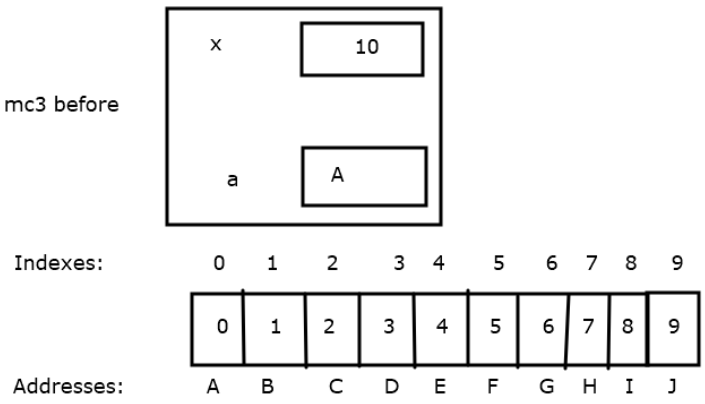
The format of the move constructor is

```
MyClass(MyClass&& myClass) noexcept;
```

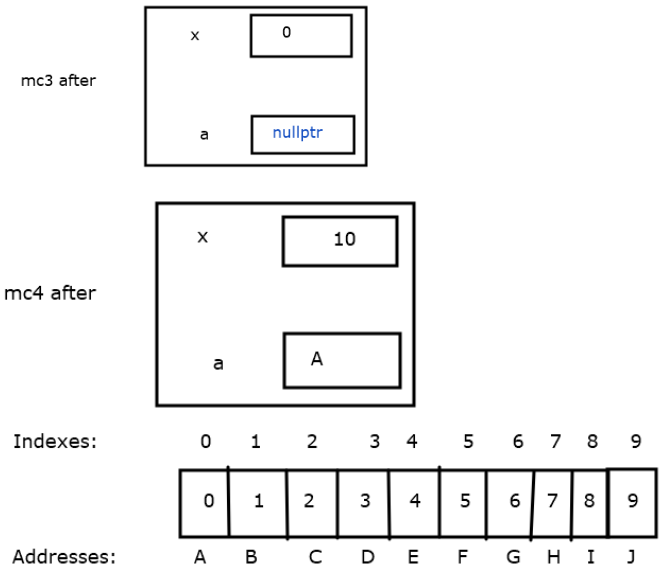
The data inside of myClass will be transferred to the calling object and not copied. The && means the function takes an rvalue but myClass is still an lvalue and <u>not</u> an rvalue. This is a common source of confusion because && means rvalue reference but myClass isn't an rvalue - it means that in the calling scope (scope where the move constructor gets called) the argument passed to it is an rvalue. However, within the scope of the function definition, myClass is an lvalue since it has a name. The noexcept keyword means the move constructor may not throw exceptions. Exception handling is a separate topic discussed in another document, but for now just know that it should be included with move constructor.

```
MyClass mc3;
MyClass mc4(move(mc3));    // move constructor -  move( ) converts mc3 to an rvalue
```

Before



After



Move Assignment Operator

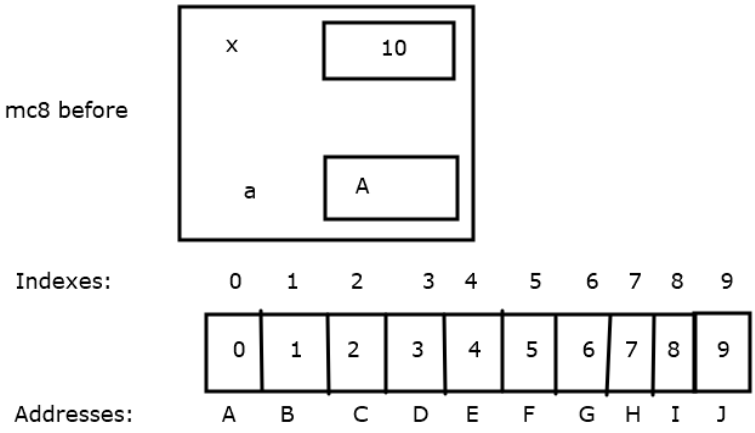
The format of the move assignment operator is:

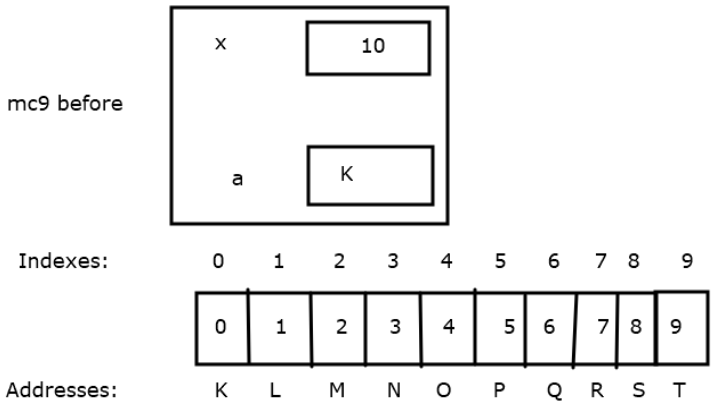
```
MyClass& operator=(MyClass&& myClass) noexcept;
```

The data in myClass will be transferred to the calling object. As with the move constructor, the && means myClass is an rvalue reference in the calling scope even though myClass is an lvalue, noexcept means the operation may not throw exceptions, and afterwards myClass must be left in a valid default state.

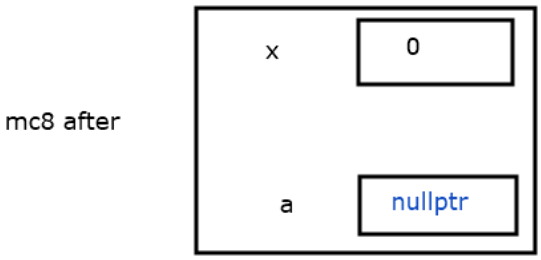
```
MyClass mc8;
MyClass mc9;
mc9 = move(mc8);            // move assignment -  move( ) converts mc8 to an rvalue
```

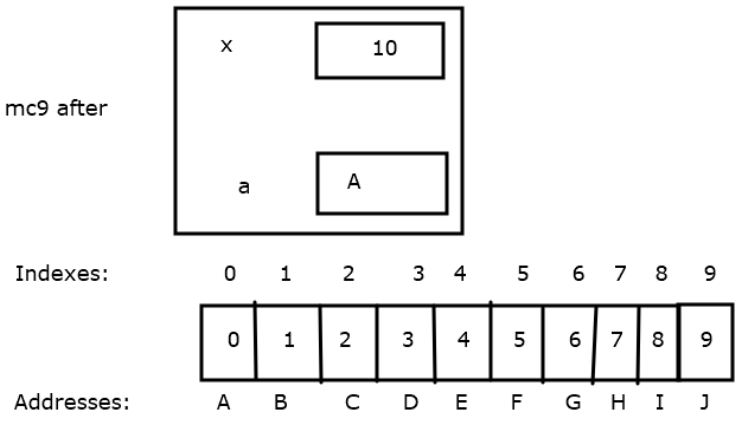
Before





After





Before transferring the content from mc8 to mc9, mc8's array (starting at address K) would need to be deallocated to prevent a memory leak.

The Rule of Three/Five/Zero

Rule of 3: If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.
Rule of 5: Because the presence of a user-defined destructor, copy-constructor, or copy-assignment operator prevents implicit definition of the move constructor and the move assignment operator, any class for which move semantics are desirable, has to declare all five special member functions:
Rule of 0: Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (which follows from the Single Responsibility Principle). Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.

These definitions can be a bit confusing so here's a more casual translation.

Rule of 3: If your class manually manages resources, you need to create a user-defined destructor, copy constructor, and copy assignment operator. For example, instead of using vector<int> for an vector of integers you decide to create it using int* and new. In this case, you need a user-defined destructor to free up the memory for the array with delete, and you need a user-defined copy constructor and copy assignment operator since the compiler generated versions will not create deep copies.

Rule of 5: If your class manually manages resources and you have user-defined versions of one of the 3 functions from the rule of three (you should have user-defined versions of all of them), you need to write a user-defined move constructor and move assignment operator if you'd like to employ move semantics.

Rule of 0: In C++, you should avoid manually managing resources in a class unless you absolutely have to. Use the built in types (vector, string etc.), and when you need to use pointers use smart pointers. All of these handle the resource management for you. So, if your class only contains these, the compiler generated destructor, copy assignment operator, copy constructor, move assignment operator, and move constructor are all satisfactory and you don't need to create your own versions. It is recommended to self-document your code that this is the case by setting them all to default which communicates this is the case. For example, let's say MyClass no longer manually managed resources and the array *a* (page 4) were changed to a vector. Using the copy constructor as an example, you would want to do the following for all 5 functions:

```
MyClass(const MyClass& myClass) = default;
```

If this weren't here, someone reading your code (or you reading your own code later) would not know that the implicitly generated versions are ok and they might think you've made a mistake by forgetting to write user-defined versions. They would have to actually further inspect your code to verify a mistake wasn't made. What this does is it communicates that you are aware that MyClass will use the compiler generated copy constructor and that you are ok with this since it doesn't manually manage resources.

Resources

Lvalues and Rvalue

- Microsoft: https://docs.microsoft.com/en-us/cpp/cpp/lvalues-and-rvalues-visual-cpp?view=msvc-160
- C++ Reference: https://en.cppreference.com/w/cpp/language/value_category

The Rule of Three/Five/Zero

- C++ Reference: https://en.cppreference.com/w/cpp/language/rule_of_three
- Article: https://www.fluentcpp.com/2019/04/23/the-rule-of-zero-zero-constructor-zero-calorie/

Copy Semantics

- Microsoft: https://docs.microsoft.com/en-us/cpp/cpp/copy-constructors-and-copy-assignment-operators-cpp?view=msvc-160
- C++ Reference
  - https://en.cppreference.com/w/cpp/language/copy_constructor
  - https://en.cppreference.com/w/cpp/language/copy_assignment

Move Semantics

- Microsoft: https://docs.microsoft.com/en-us/cpp/cpp/move-constructors-and-move-assignment-operators-cpp?view=msvc-160
- C++ Reference
  - https://en.cppreference.com/w/cpp/language/move_constructor
  - https://en.cppreference.com/w/cpp/language/move_assignment

ISO C++ Core Guidelines

Main Website: https://isocpp.org/
Core Guidelines: http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines