

## 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 this document, 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 *ISO 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

<b><u>Topics</u></b>	<b><u>Page</u></b>
Lvalues and Rvalues	2
The Rule of Three/Five/Zero	2
Copy Semantics	3
Move Semantics	5
Resources	9

## 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 hardcoded 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.

A common misconception about lvalues and rvalues is that the l and r stand for "left" and "right" meaning lvalues go on the left side of the = operator, and r values go on the right side. As seen in the example below, this is not the case

<code>int x = 3;</code>	x is an <i>lvalue</i> , 3 is an <i>rvalue</i>
<code>int y = x;</code>	x and y are both <i>lvalues</i>
<code>string s = "Liebestraum";</code>	s is an <i>lvalue</i> , "Liebestraum" is an <i>rvalue</i>
<code>string s1 = string("Liebestraum");</code>	s1 is an <i>lvalue</i> , <code>string("Liebestraum")</code> is an <i>rvalue</i>
<code>string s2 = s;</code>	s2 and s are both <i>lvalues</i>

An rvalue can in fact go on the left side of the = operator.

<code>s + s = s;</code>	s + s is an <i>rvalue</i> , s is an <i>lvalue</i>
-------------------------	---

Though this statement has zero utility, it is still valid C++ code.

There ARE cases where you can refer to an rvalue by a name. This is when **&&**, which is the **rvalue reference declarator**, is used. Similarly, **&** is the **lvalue reference declarator**. The use of **&&** is most commonly seen in move semantics.

## 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

## 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;
```

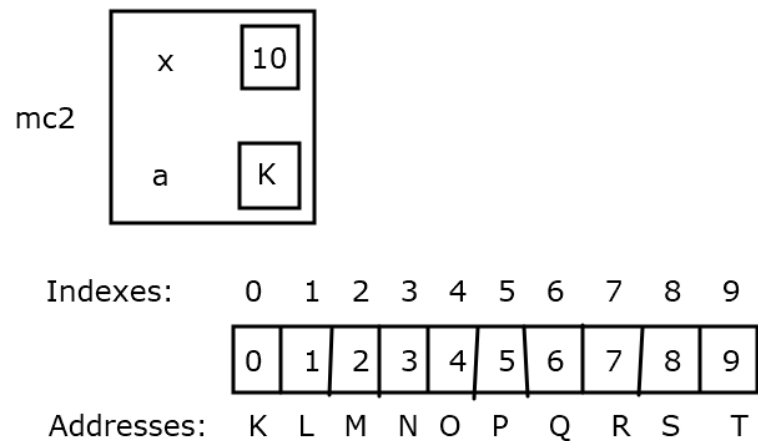
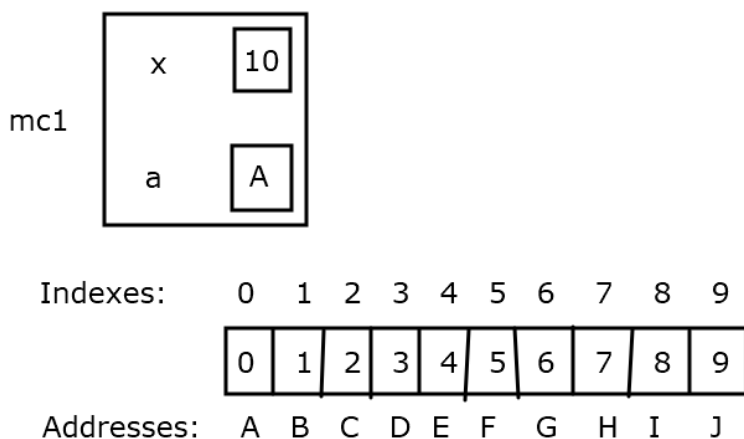
the format of the copy constructor is as follows

```
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 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.

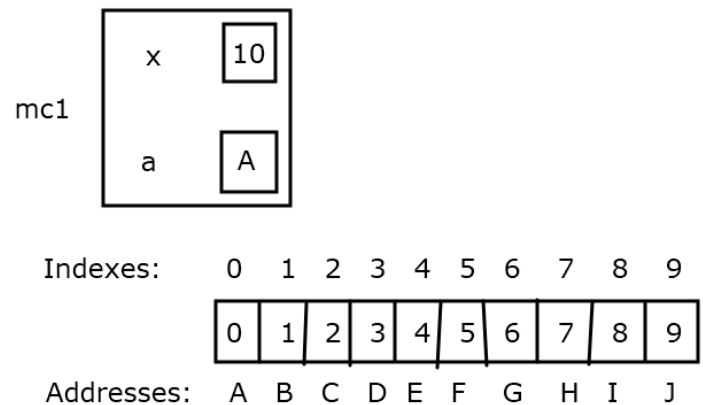
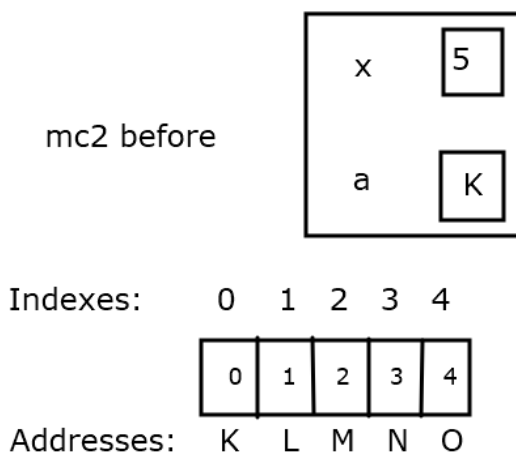
```
MyClass mc1;
```

```
MyClass mc2(5);
```

```
mc2 = mc1;
```

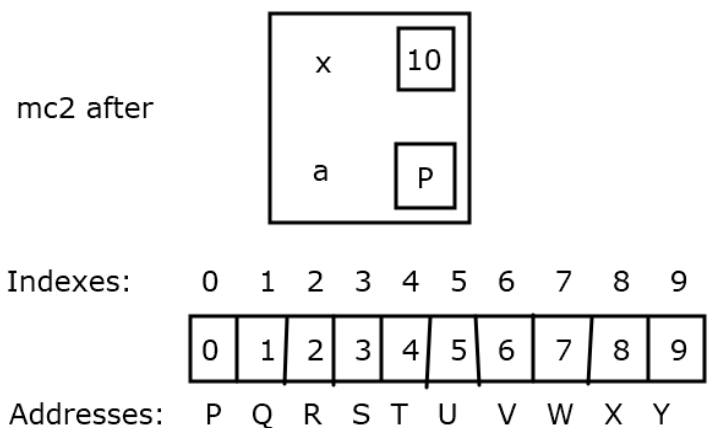
// copy assignment operator on an existing object

*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 it held a lot of 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;
```

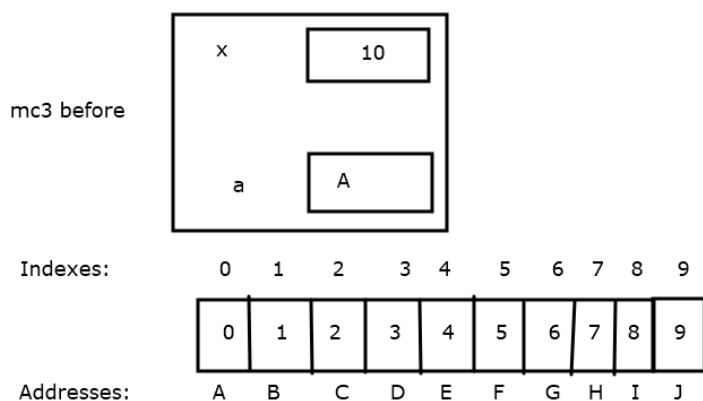
The data inside of `myClass` will be transferred to the calling object and not copied.

The `&&` means `myClass` is an rvalue reference, and 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.

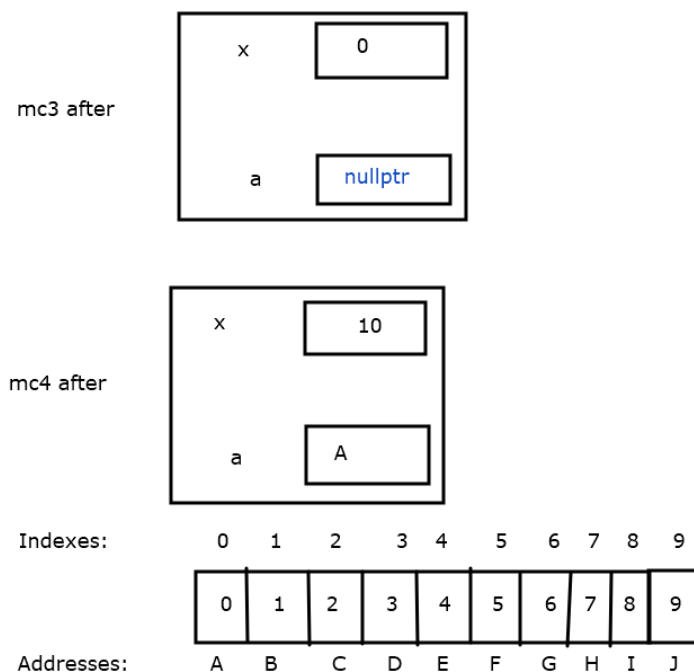
```
MyClass mc3;
```

```
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;
```

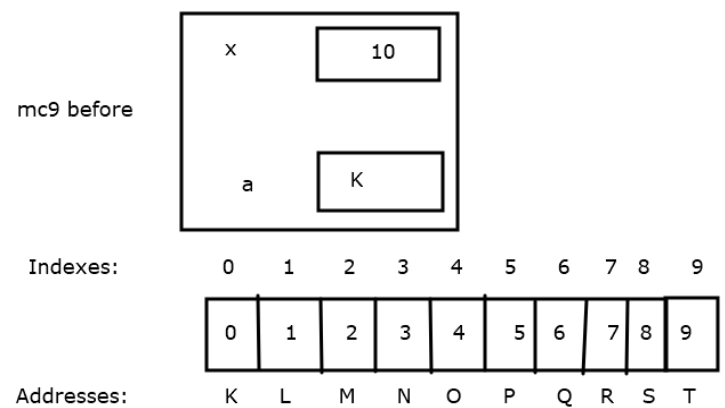
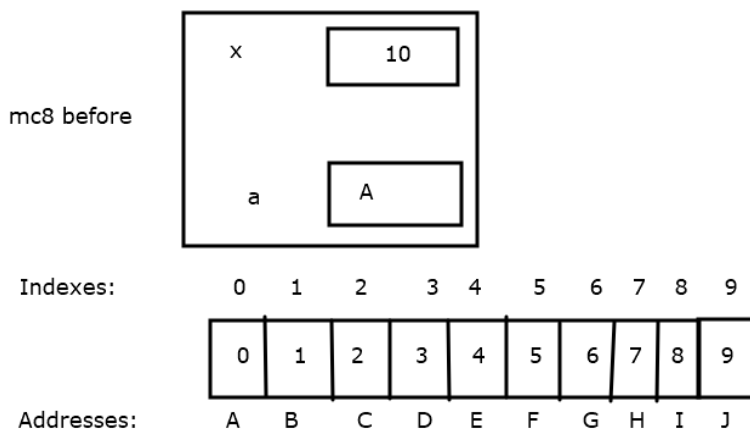
The data in myClass will be transferred to the calling object. As with the move constructor, the && means myClass is an rvalue reference, noexcept means the operation may not throw exceptions, and afterwards myClass must be left in a valid default state.

```
MyClass mc8;
```

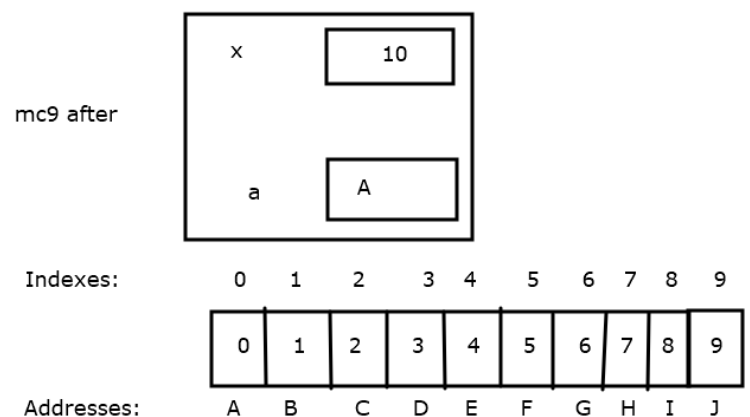
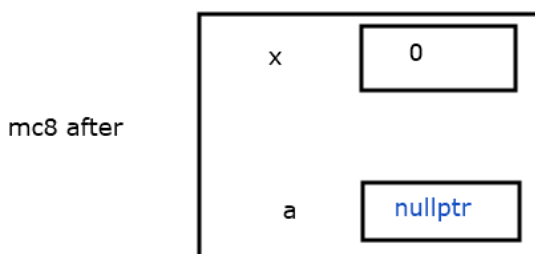
```
MyClass mc9;
```

```
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.

## 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](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](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_constructor)
  - [https://en.cppreference.com/w/cpp/language/copy\\_assignment](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_constructor)
  - [https://en.cppreference.com/w/cpp/language/move\\_assignment](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>