Introduction

This document provides a basic introduction to C++ copy and move semantics. It should be used in conjunction with the example code files. There are additional important details about copy and move semantics not discussed because this is a basic introduction. The official online resources for C++ can be consulted for these details. Links to some of these resources, specifically Microsoft and C++ Reference, are provided at the end in the **Resources** section. Both are excellent resources, and the Microsoft links are more beginner friendly and easier to understand.

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

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

`int x = 3;`	x is an *lvalue*, 3 is an *rvalue*
`int y = x;`	x and y 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*

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

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

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.

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. 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 internal representation

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

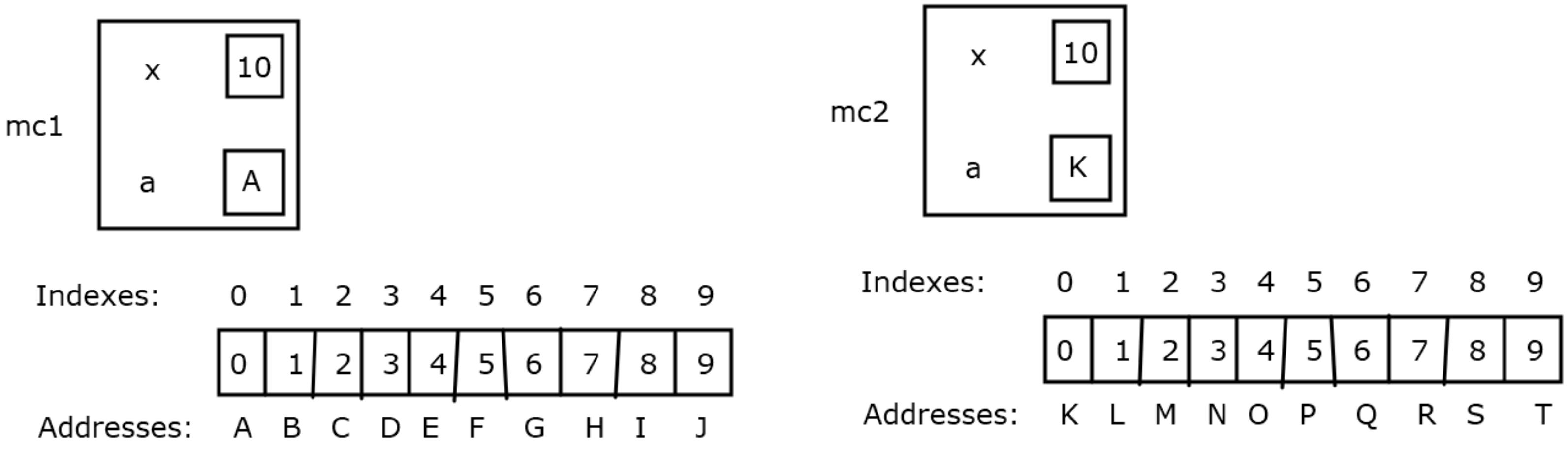
the format of the copy constructor is as follows

```
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
```

In the first example of the copy constructor. An object named mc1 is created. Then in the next line, a new object named mc2 is created that is a complete and independent copy of mc1.


mc1
x
10
a
A
Indexes: 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
Addresses: A B C D E F G H I J
mc2
x
10
a
K
Indexes: 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
Addresses: K L M N O P Q R S T

In the above diagram, note how mc2 is a complete and independent copy of mc1. They don't share the same array - mc1 has one array, and mc2 has a completely separate array with the same capacity and values stored in the indexes. 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

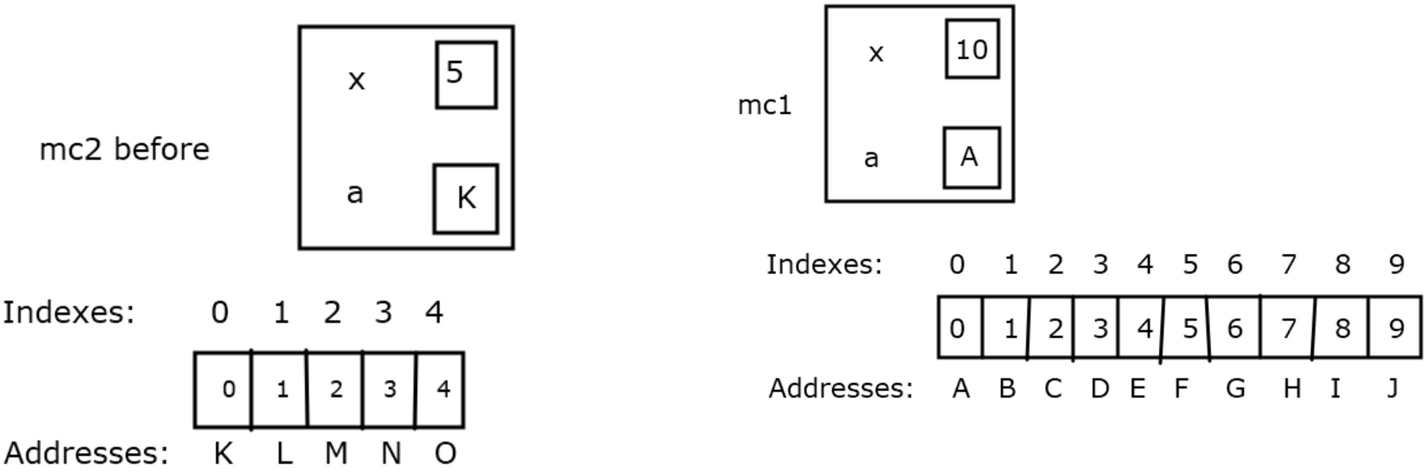
For MyClass, the format of the copy assignment operator is

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

Again, This means that a complete and independent copy of myClass is going to be created and stored in either the new object being instantiated, or an already existing object. Once again, 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.

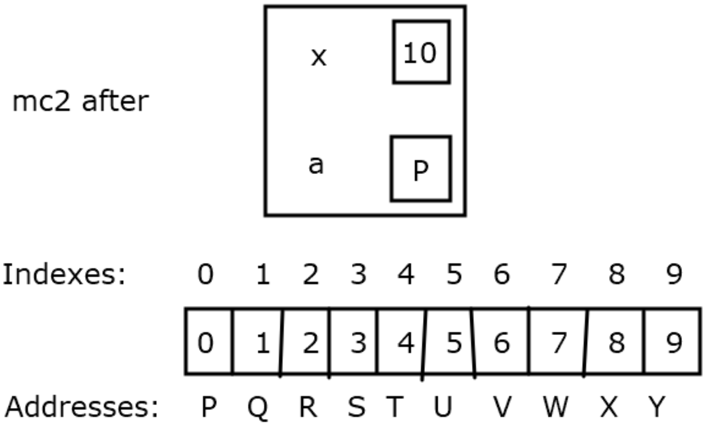
```
        MyClass mc1;
        MyClass mc2(5);
        mc2 = mc1;                    // copy assignment operator on an existing object
```

Before Copy Assignment Operator



After Copy Assignment Operator

As with the copy constructor, mc2 is now a complete and independent 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.


mc2 after
x
10
a
P
Indexes: 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
Addresses: P Q R S T U V W X Y