Copying Classes

There are many times we want to make a copy of something in C++; an assignment statement, passing a parameter, or creating a new instance of something.

Copying simple data types

If we were working with a simple data type like an integer, we copy it very simply.

```
int a = 5;
int b;

b = a;
```

Copying a simple class

When we have a class that contains simple data types, it works just the same:

```
class simple{
private:
    int value;
public:
    simple(int value  = 0) { this->value = value; }
   void setValue(int value) { this->value = value;}
   int getValue() { return this->value; }
};

int main()
{
    simple object1(4);
    simple object2(6);

        // display initial values of 4 and 6
    cout << object1.getValue() << " " << object2.getValue() << endl;

        // copy object1 to object2 and show they are both 4
    object2 = object1;
    cout << object1.getValue() << " " << object2.getValue() << endl;

        // change object 2 and see that they are now 4 and 7
    object2.setValue(7);
    cout << object1.getValue() << " " << object2.getValue() << endl;

    return 0;
}
```

The resulting output is exactly what you would expect:

```
4 6
4 4
4 7
Program ended with exit code: 0
```

Copying a more complex class

Now, consider the situation when you have a pointer to an integer and the integer itself is on the stack.

```
class complex{
private:
    int * value;
public:
    complex(int value = 0) { this->value = new int(value); }
    void setValue(int value) { *this->value = value; }
    int getValue() { return *this->value; }
};

int main()
{
    complex object1(4);
    complex object2(6);

        // display initial values of 4 and 6
    cout << object1.getValue() << " " << object2.getValue() << endl;

        // copy object1 to object2 and show they are both 4
    object2 = object1;
    cout << object1.getValue() << " " << object2.getValue() << endl;

        // change object 2 and see that they are now 4 and 7
    object2.setValue(7);
    cout << object1.getValue() << " " << object2.getValue() << endl;


    return 0;
}
```

You would expect the same output as before, but now it is:

```
4 6
4 4
7 7
Program ended with exit code: 0
```

Lets add some more output and see what is really happening. Look at the addresses that are stored in value:

```
class complex{
private:
    int * value;
public:
    complex(int value = 0) { this->value = new int(value); }
    void setValue(int value) { *this->value = value; }
    int getValue() { return *this->value; }
    int* getAddress() { return value; }
};

int main()
{
    complex object1(4);
    complex object2(6);

        // display initial values of 4 and 6
        // also display the addresses where these values are
    cout << object1.getValue() << " " << object2.getValue() << endl;
    cout << object1.getAddress() << " " << object2.getAddress() << endl;

        // copy object1 to object2 and show they are both 4
        // and that the addresses are now the same
    object2 = object1;
    cout << object1.getValue() << " " << object2.getValue() << endl;
    cout << object1.getAddress() << " " << object2.getAddress() << endl;

        // change object 2 and see that they are now both 7 and 7
        // since they point to the same location, changing one changes both
    object2.setValue(7);
    cout << object1.getValue() << " " << object2.getValue() << endl;
    cout << object1.getAddress() << " " << object2.getAddress() << endl;


    return 0;
}
```

And the output shows that the assignment statement set the addresses to the same, not the contents of those addresses:

```
4 6
0x10078c460 0x100787830
4 4
0x10078c460 0x10078c460
7 7
0x10078c460 0x10078c460
Program ended with exit code: 0
```

Explanation (shallow copy)

When you copy an object, what is done by default is a **shallow** copy. This means that each location in the first object is copied bit-by-bit to the second object. If they initially pointed to two different locations in memory, then they now point to the same location, since the **address** itself was overwritten.

Solution (deep copy)

The solution to the problem is to perform what is known as a **deep** copy. This is when the contents of the location being pointed to copied from one to the other, rather than the addresses. Since this is not what the compiler does by default, you have to explicitly tell it to do so. You do this by defining a new assignment operator or a copy constructor.

Overloaded assignment

In C++, you can define a new behavior for an existing operator. Since this is the same as creating another function with a new parameter list, it is known as overloading the operator. The most common operator to overload is the assignment operator (=), since it is required anytime you have a class that contains pointers.

See the new operator definition in our revised complex class. If you think about the assignment, it is a binary operator with a left-hand-side and a right-hand-side. When you define an operator in a class, the object being called is treated as the left-hand-side of the operator, so you need to pass in the address of the right hand side. You should pass this in using a constant reference to avoid the extra overhead of copying. You also need to return the address of the current object so that you can string assignment statements:

```
A = B = C;
```

Here is the revised version of our above program (note that the getValue is now defined a const function so that it can be called from a const reference:

```
class complex{
private:
    int * value;
public:
    complex(int value = 0) { this->value = new int(value); }
    void setValue(int value) { *this->value = value; }
    int getValue() const { return *this->value; }
    int* getAddress() { return value; }

        // overloaded assignment operator
    complex & operator=( const complex & rhs )
    {
            // copy the contents of the location pointed to by value
        *(this->value) = *rhs.value;

            // return our address
        return *this;
    }
};
```

```
int main()
{
    complex object1(4);
    complex object2(6);

        // display initial values of 4 and 6
        // also display the addresses where these values are
    cout << object1.getValue() << " " << object2.getValue() << endl;
    cout << object1.getAddress() << " " << object2.getAddress() << endl;

        // copy object1 to object2 and show they are both 4
        // and that the addresses are now the same
    object2 = object1;
    cout << object1.getValue() << " " << object2.getValue() << endl;
    cout << object1.getAddress() << " " << object2.getAddress() << endl;

        // change object 2 and see that they are now both 7 and 7
        // since they point to the same location, changing one changes both
    object2.setValue(7);
    cout << object1.getValue() << " " << object2.getValue() << endl;
    cout << object1.getAddress() << " " << object2.getAddress() << endl;


    return 0;
}
```

And the output shows that the assignment statement did not change the addresses, only the contents of those addresses:

```
4 6
0x102101c60 0x102101ce0
4 4
0x102101c60 0x102101ce0
4 7
0x102101c60 0x102101ce0
Program ended with exit code: 0
```

Copy Constructor

We create a copy constructor for a similar reason. Sometimes we want to create a new object that starts with the same values as an existing object, but they both contain pointers. It is just like our other constructors, just has a different parameter type. See the example below:

```
class complex{
private:
    int * value;
public:
    complex(int value = 0) { this->value = new int(value); }
    complex( const complex & other ) { this->value = new int(*other.value); }

    void setValue(int value) { *this->value = value; }
    int getValue() const { return *this->value; }
    int* getAddress() { return value; }

        // overloaded assignment operator
    complex & operator=( const complex & rhs )
    {
            // copy the contents of the location pointed to by value
        *(this->value) = *rhs.value;

            // return our address
        return *this;
    }
};
```