Deconstructing the Assignment Operator

A detailed examination of each facet of the code used in overloading the assignment operator gives useful insights into several features of C++. Consider the following code

```
#include <iostream>

using namespace std;

class aClass
{
    public:
        int data;

        aClass() {
            cout << "calling default constructor\n";
        }

        aClass(int x): data(x) {
            cout << "calling int constructor\n";
        }

        aClass(const aClass &other) {
            cout << "calling copy constructor\n";
            data = other.data;
        }

        ~aClass() {
            cout << "calling destructor\n";
        }

        aClass& operator = (const aClass &other) {
            cout << "calling assignment operator\n";
            data = other.data;
            return *this;
        }
};

int main()
{
    aClass ob1, ob2, ob3(3);

    cout << "\nStart using assignment\n";
    ob1 = ob2 = ob3;
    cout << "Finish using assignment\n";
    cout << "ob1.data = " << ob1.data << "\n\n";

    return 0;
}
```

As you can see, `aClass` declares some constructors including a copy constructor. We have also declared the destructor plus the assignment operator has been overload. In each of these, whenever they are invoked they will print a message.

In `main`, we simply declare 3 `aClass` objects, invoke the assignment operator twice, print the result and finish up. Here is the result when we compile and run the code.

```
calling default constructor
calling default constructor
calling int constructor

Start using assignment
calling assignment operator
calling assignment operator
Finish using assignment
ob1.data = 3

calling destructor
calling destructor
calling destructor
```

The first 3 lines tell us the constructors are invoked when we declare the 3 objects. The int constructor is called when we declare `ob3(3)`.

In the assignment line, the assignment operator is invoked twice. And the result in `ob1.data` is correctly printed.

Finally, the destructor for each object is invoked as `main` finishes up.

Make sure you understand this code and its output thoroughly before continuing on.

Returning `aClass` instead of `aClass&`

Now, let us remove the first & from the assignment operator code. That is

```
aClass operator = (const aClass &other) {
    cout << "calling assignment operator\n";
    data = other.data;
    return *this;
}
```

Instead of returning a reference to the `*this` object, we will be returning a copy. When compiled and run this is what happens

```
calling default constructor
calling default constructor
calling int constructor

Start using assignment
calling assignment operator
calling copy constructor
calling assignment operator
calling copy constructor
calling destructor
calling destructor
Finish using assignment
ob1.data = 3

calling destructor
calling destructor
calling destructor
```

Note that the final result of invoking the assignment operator is a call to the copy constructor. Furthermore, this creates an extra two objects that have to be destroyed when the assignment line is finished.

This is a lot of unnecessary operations to perform.

Why return `aClass&` at all? Why not return `void`?

Ok, let's try it.

```
void operator = (const aClass other) {
    cout << "calling assignment operator\n";
    data = other.data;
}
```

When I try and compile I get the following error message.

```
assigncode.cpp: In function 'int main()':
assigncode.cpp:38: error: no match for 'operator=' in 'ob1 = ob2.
aClass::operator=(((const aClass&)((const aClass*)(& ob3))))'
assigncode.cpp:27: note: candidates are: void aClass::operator=(const aClass&)
```

It's complaining about the `ob1 = ob2`. If I change the code in `main` from `ob1 = ob2 = ob3` to ob1 = ob3 it compiles fine.

What we are really doing is making `ob1` equal to the result from `ob2 = ob3`. However, if we are returning a `void` then it doesn't make sense to make `ob1 = void`. Hopefully you can see the reason for returning the class reference.

What about the reference in `(const aClass &other)`?
That is

```
aClass& operator = (const aClass other) {
    cout << "calling assignment operator\n";
    data = other.data;
    return *this;
}
```

When we run this we get the following

```
calling default constructor
calling default constructor
calling int constructor

Start using assignment
calling copy constructor
calling assignment operator
calling assignment operator
calling destructor
Finish using assignment
Ob1.data = 3

calling destructor
```

```
calling destructor
calling destructor
```

We are sending a copy of the object when we first invoke the assignment declaration. When we talk about a copy it means invoking the copy constructor. This also means another object to destroy when finished. More unnecessary operations.

What happens if we remove the `const`?

```
aClass operator = (aClass &other) {
    cout << "calling assignment operator\n";
    data = other.data;
    return *this;
}
```

Actually, nothing at all. The code compiles and runs in exactly the same way as it initially ran. So why bother? Well, let's change the assignment code to the following.

```
aClass operator = (aClass &other) {
    cout << "calling assignment operator\n";
    data = other.data;
    other.data = 0;
    return *this;
}
```

I've now added a line to change the `data` component of `other`. This is perfectly legal and the code will compile without any problems.

Let's think about this for a moment. I have used `aClass &other` in my parameter. In this case, having `other.data = 0;` will change the contents of the object in `main` that is being called in the assignment operation.

Do we really want this to happen? In some functions the answer is yes. In this case however, the answer is definitely NO. All the assignment operation should be doing is duplicating the contents of `other` in `*this` but NOT changing it.

Including the use of `const` guarantees the compiler will catch this sort of thing happening and return a compiler error. It is ALWAYS better to set things up so the compiler catches things rather than forming subtle logic errors. See what happens when I add `const` back in and try to compile.

```
assigncode.cpp: In member function 'aClass aClass::operator=(aClass)':
assigncode.cpp:30: error: assignment of data-member 'aClass::data' in read-only
structure
```

I hope in deconstructing the assignment operation you have gained some understanding about why it is constructed the way it is.