9.14 — Overloading the assignment operator

BY ALEX ON JUNE 5TH, 2016 | LAST MODIFIED BY ALEX ON MAY 19TH, 2018

The **assignment operator** (operator=) is used to copy values from one object to another *already existing object*.

Assignment vs Copy constructor

The purpose of the copy constructor and the assignment operator are almost equivalent -- both copy one object to another. However, the copy constructor initializes new objects, whereas the assignment operator replaces the contents of existing objects.

The difference between the copy constructor and the assignment operator causes a lot of confusion for new programmers, but it's really not all that difficult. Summarizing:

- If a new object has to be created before the copying can occur, the copy constructor is used (note: this includes passing or returning objects by value).
- If a new object does not have to be created before the copying can occur, the assignment operator is used.

Overloading the assignment operator

Overloading the assignment operator (operator=) is fairly straightforward, with one specific caveat that we'll get to. The assignment operator must be overloaded as a member function.

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    // Copy constructor
    Fraction(const Fraction &copy) :
        m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
    {
        // no need to check for a denominator of 0 here since copy must already be a valid Fraction
        std::cout << "Copy constructor called\n"; // just to prove it works
    }

        // Overloaded assignment
        Fraction& operator= (const Fraction &fraction);

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);

};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

// A simplistic implementation of operator= (see better implementation below)
Fraction& Fraction::operator= (const Fraction &fraction)
{
```

```
    // do the copy
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}

int main()
{
    Fraction fiveThirds(5, 3);
    Fraction f;
    f = fiveThirds; // calls overloaded assignment
    std::cout << f;

    return 0;
}
```

This prints:

```
5/3
```

This should all be pretty straightforward by now. Our overloaded operator= returns *this, so that we can chain multiple assignments together:

```
int main()
{
    Fraction f1(5,3);
    Fraction f2(7,2);
    Fraction f3(9,5);

    f1 = f2 = f3; // chained assignment

    return 0;
}
```

Issues due to self-assignment

Here's where things start to get a little more interesting. C++ allows self-assignment:

```
int main()
{
    Fraction f1(5,3);
    f1 = f1; // self assignment

    return 0;
}
```

This will call f1.operator=(f1), and under the simplistic implementation above, all of the members will be assigned to themselves. In this particular example, the self-assignment causes each member to be assigned to itself, which has no overall impact, other than wasting time. In most cases, a self-assignment doesn't need to do anything at all!

However, in cases where an assignment operator needs to dynamically assign memory, self-assignment can actually be dangerous:

```
#include <iostream>

class MyString
{
private:
    char *m_data;
    int m_length;

public:
    MyString(const char *data="", int length=0) :
        m_length(length)
    {
```

```
        if (!length)
            m_data = nullptr;
        else
            m_data = new char[length];

        for (int i=0; i < length; ++i)
            m_data[i] = data[i];
    }

    // Overloaded assignment
    MyString& operator= (const MyString &str);

    friend std::ostream& operator<<(std::ostream& out, const MyString &s);
};

std::ostream& operator<<(std::ostream& out, const MyString &s)
{
    out << s.m_data;
    return out;
}

// A simplistic implementation of operator= (do not use)
MyString& MyString::operator= (const MyString &str)
{
    // if data exists in the current string, delete it
    if (m_data) delete[] m_data;

    m_length = str.m_length;

    // copy the data from str to the implicit object
    m_data = new char[str.m_length];

    for (int i=0; i < str.m_length; ++i)
        m_data[i] = str.m_data[i];

    // return the existing object so we can chain this operator
    return *this;
}

int main()
{
    MyString alex("Alex", 5); // Meet Alex
    MyString employee;
    employee = alex; // Alex is our newest employee
    std::cout << employee; // Say your name, employee

    return 0;
}
```

First, run the program as it is. You'll see that the program prints "Alex" as it should.

Now run the following program:

```
int main()
{
    MyString alex("Alex", 5); // Meet Alex
    alex = alex; // Alex is himself
    std::cout << alex; // Say your name, Alex

    return 0;
}
```

You'll probably get garbage output. What happened?

Consider what happens in the overloaded operator= when the implicit object AND the passed in parameter (str) are both variable alex. In this case, m_data is the same as str._m_data. The first thing that happens is that the function checks to see if the implicit object

already has a string. If so, it needs to delete it, so we don't end up with a memory leak. In this case, m_data is allocated, so the function deletes m_data. But str.m_data is pointing to the same address! This means that str.m_data is now a dangling pointer.

Later on, we allocate new memory to m_data (and str.m_data). So when we subsequently copy the data from str.m_data into m_data, we're copying garbage, because str.m_data was never initialized.

Detecting and handling self-assignment

Fortunately, we can detect when self-assignment occurs. Here's a better implementation of our overloaded operator= for the Fraction class:

```
// A better implementation of operator=
Fraction& Fraction::operator= (const Fraction &fraction)
{
    // self-assignment guard
    if (this == &fraction)
        return *this;

    // do the copy
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}
```

By checking if our implicit object is the same as the one being passed in as a parameter, we can have our assignment operator just return immediately without doing any other work.

Note that there is no need to check for self-assignment in a copy-constructor. This is because the copy constructor is only called when new objects are being constructed, and there is no way to assign a newly created object to itself in a way that calls to copy constructor.

Default assignment operator

Unlike other operators, the compiler will provide a default public assignment operator for your class if you do not provide one. This assignment operator does memberwise assignment (which is essentially the same as the memberwise initialization that default copy constructors do).

Just like other constructors and operators, you can prevent assignments from being made by making your assignment operator private or using the delete keyword:

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    // Copy constructor
    Fraction(const Fraction &copy) = delete;

    // Overloaded assignment
    Fraction& operator= (const Fraction &fraction) = delete; // no copies through assignment!

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);

```

```
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

int main()
{
    Fraction fiveThirds(5, 3);
    Fraction f;
    f = fiveThirds; // compile error, operator= has been deleted
    std::cout << f;

    return 0;
}
```

9.15 -- Shallow vs. deep copying

Index

9.13 -- Converting constructors, explicit, and delete

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

55 comments to 9.14 — Overloading the assignment operator

Fortunato Rosa
June 19, 2018 at 11:25 am · Reply

Hello Alex, first of all I want to compliment with you about this C++ course which is fantastic and very clear. I have a question about the "=" operator overload: