9.14 — Overloading the assignment operator

BY ALEX ON JUNE 5TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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 Fr
action
        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;
```

```
60 }
```

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 an updated implementation of our overloaded operator= for the MyString class:

```
// A simplistic implementation of operator= (do not use)
MyString& MyString::operator= (const MyString& str)
{
    // self-assignment check
    if (this == &str)
        return *this;

    // 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;
}
```

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

Because this is just a pointer comparison, it should be fast, and does not require operator== to be overloaded.

When not to handle self-assignment

First, 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.

Second, the self-assignment check may be omitted in classes that can naturally handle self-assignment. Consider this Fraction class assignment operator that has a self-assignment guard:

```
// 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; // can handle self-assignment
    m_denominator = fraction.m_denominator; // can handle self-assignment

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

If the self-assignment guard did not exist, this function would still operate correctly during a self-assignment (because all of the operations done by the function can handle self-assignment properly).

Because self-assignment is a rare event, some prominent C++ gurus recommend omitting the self-assignment guard even in classes that would benefit from it. We do not recommend this, as we believe it's a better practice to code defensively and then selectively optimize later.

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);

};
```

```
27
28  std::ostream& operator<<(std::ostream& out, const Fraction &f1)
29  {
30      out << f1.m_numerator << "/" << f1.m_denominator;
31      return out;
32  }
33
34  int main()
35  {
36      Fraction fiveThirds(5, 3);
37      Fraction f;
38      f = fiveThirds; // compile error, operator= has been deleted
39      std::cout << f;
40
41      return 0;
42  }
```

9.15 -- Shallow vs. deep copying

Index

9.13 -- Converting constructors, explicit, and delete

C++ TUTORIAL | PRINT THIS POST

108 comments to 9.14 — Overloading the assignment operator

« Older Comments 1 2

Sid22
January 25, 2020 at 5:00 am · Reply

In the book I am reading, it is written about overloaded assignment operator with its return type as constant reference :

```
1  const Array& Array::operator=(const Array& right)
```

(where Array is a user defined class implementing dynamic array)
It says:
" Regardless of whether this is a self-assignment, the member function (talking about the assignment operator overloading function) returns the current object (i.e., *this) as a constant reference; this enables cascaded Array assignments such as x = y = z, but prevents ones like (x = y) = z because z cannot be assigned to the const Array reference that's returned by (x = y). "

-> Why is (x=y)=z any different than x=y=z?
->What is the difference between

```
1  const Array& Array::operator=(const Array& right)
```

and

```
Array& Array::operator=(const Array& right)
```

nascardriver
January 25, 2020 at 5:18 am · Reply

The assignment operator is evaluated right-to-left, ie.

```
a = b = c = d = e
// evaluated as
(a = (b = (c = (d = e))))
```

Here, it doesn't matter whether `(d = e)` is `const` or not, because it doesn't get modified. In `(x = y) = z`, we're trying to modify a `const` reference, which doesn't work.

I can't tell which version is better.

Sid22
January 27, 2020 at 7:04 am · Reply

Thanks! Cleared my doubt!

David
January 23, 2020 at 12:13 am · Reply

I have a question in the following code.As you mentoned,"If a new object has to be created before the copying can occur, the copy constructor is used",so why is the copy constructor not used ? We have to create a new object named f before we copy the contents of the object named fiveThird into the object named f.

I think this behavior meets your description for the copy constructor.Does my concept be wrong?

```
 #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;
}
```

sito
February 3, 2020 at 4:52 am · Reply

alright. So I think i have an answer. I don't know if this will answer your question so Alex or Nascardriver feel free to corret me if i'm wrong.
The assignement operator copys and replases the content of the object with new values that are being assigned. In order for this to work though the object have to exist otherwise you can't replase anything since there is not an existing object to plase the copied content from the copied object in. So if you do something like this, Fraction f3=f1 the compiler will check if f3 already exists. The compiler will then determine that f3 have not been created before so assigning a value to it won't work. The compiler will instead create f3 and use the copy constructor.
You can test this to see what I mean.
Sorry if this is more rambling then a good answer but the best way to understand the diference is to test your code.

Victor
November 21, 2019 at 12:34 pm · Reply

Inside the assignment operator of MyString, do I really need to delete[] m_data? Reusing the allocated memory would lead to problems? If instead of deleting and allocating I do something like:

```
if(!m_data) m_data = new char[str.m_length];
//...copy the values
```

Would it be wrong?

nascardriver
November 22, 2019 at 7:28 am · Reply

If the new string is not longer than the old string, you can (and it's good to) re-use the memory.

« Older Comments 1 2