21.12 — Overloading the assignment operator

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

Related content

As of C++11, C++ also supports “Move assignment”. We discuss move assignment in lesson 22.3 -- Move constructors and move assignment.

Copy assignment vs Copy constructor

The purpose of the copy constructor and the copy 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 copy 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 copy assignment operator (operator=) is fairly straightforward, with one specific caveat that we’ll get to. The copy assignment operator must be overloaded as a member function.

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

class Fraction
{
private:
	int m_numerator { 0 };
	int m_denominator { 1 };

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 <algorithm> // for std::max and std::copy_n
#include <iostream>

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

public:
	MyString(const char* data = nullptr, int length = 0 )
		: m_length { std::max(length, 0) }
	{
		if (length)
		{
			m_data = new char[static_cast<std::size_t>(length)];
			std::copy_n(data, length, m_data); // copy length elements of
data into m_data
		}
	}
	~MyString()
	{
		delete[] m_data;
	}

	// 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;
	m_data = nullptr;

	// allocate a new array of the appropriate length
	if (m_length)
		m_data = new char[static_cast<std::size_t>(str.m_length)];

	std::copy_n(str.m_data, m_length, m_data); // copies m_length elements of
str.m_data into m_data
```

```
        // 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 because str is the same as *this, the string that we wanted to copy has been deleted and m_data (and str.m_data) are dangling.

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:

```
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;
        m_data = nullptr;

        // allocate a new array of the appropriate length
        if (m_length)
                m_data = new char[static_cast<std::size_t>(str.m_length)];

        std::copy_n(str.m_data, m_length, m_data); // copies m_length elements of
str.m_data into m_data

        // 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

Typically the self-assignment check is skipped for copy constructors. Because the object being copy constructed is newly created, the only case where the newly created object can be equal to the object being copied is when you try to initialize a newly defined object with itself:

```
someClass c { c };
```

In such cases, your compiler should warn you that `c` is an uninitialized variable.

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.

The copy and swap idiom

A better way to handle self-assignment issues is via what's called the copy and swap idiom. There's a great writeup of how this idiom works on Stack Overflow.

The implicit copy assignment operator

Unlike other operators, the compiler will provide an implicit public copy assignment operator for your class if you do not provide a user-defined 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 copy assignment operator private or using the delete keyword:

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

class Fraction
{
private:
        int m_numerator { 0 };
        int m_denominator { 1 };

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;
}
```

Note that if your class has const members, the compiler will instead define the implicit `operator=` as deleted. This is because const members can't be assigned, so the compiler will assume your class should not be assignable.

If you want a class with const members to be assignable (for all members that aren't const), you will need to explicitly overload `operator=` and manually assign each non-const member.