9.x — Chapter 9 comprehensive quiz

BY ALEX ON AUGUST 9TH, 2016 | LAST MODIFIED BY ALEX ON JULY 17TH, 2018

In this chapter, we explored topics related to operator overloading, as well as overloaded typecasts, and topics related to the copy constructor.

Summary

Operator overloading is a variant of function overloading that lets you overload operators for your classes. When operators are overloaded, the intent of the operators should be kept as close to the original intent of the operators as possible. If the meaning of an operator when applied to a custom class is not clear and intuitive, use a named function instead.

Operators can be overloaded as a normal function, a friend function, or a member function. The following rules of thumb can help you determine which form is best for a given situation:

- If you're overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function.
- If you're overloading a unary operator, do so as a member function.
- If you're overloading a binary operator that modifies its left operand (e.g. operator+=), do so as a member function if you can.
- If you're overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function or friend function.

Typecasts can be overloaded to provide conversion functions, which can be used to explicitly or implicitly convert your class into another type.

A copy constructor is a special type of constructor used to initialize an object from another object of the same type. Copy constructors are used for direct/uniform initialization from an object of the same type, copy initialization (Fraction f = Fraction(5,3)), and when passing or returning a parameter by value.

If you do not supply a copy constructor, the compiler will create one for you. Compiler-provided copy constructors will use memberwise initialization, meaning each member of the copy is initialized from the original member. The copy constructor may be elided for optimization purposes, even if it has side-effects, so do not rely on your copy constructor actually executing.

Constructors are considered converting constructors by default, meaning that the compiler will use them to implicitly convert objects of other types into objects of your class. You can avoid this by using the explicit keyword in front of your constructor. You can also delete functions within your class, including the copy constructor and overloaded assignment operator if desired. This will cause a compiler error if a deleted function would be called.

The assignment operator can be overloaded to allow assignment to your class. If you do not provide an overloaded assignment operator, the compiler will create one for you. Overloaded assignment operators should always include a self-assignment check.

New programmers often mix up when the assignment operator vs copy constructor are used, but it's fairly straightforward:

- 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.

By default, the copy constructor and assignment operators provided by the compiler do a memberwise initialization or assignment, which is a shallow copy. If your class dynamically allocates memory, this will likely lead to problems, as multiple objects will end up pointing to the same allocated memory. In this case, you'll need to explicitly define these in order to do a deep copy. Even better, avoid doing your own memory management if you can and use classes from the standard library.

Quiz Time

1) Assuming Point is a class and point is an instance of that class, should you use a normal/friend or member function overload for the following operators?

1a) point + point
1b) -point
1c) std::cout << point
1d) point = 5;

Hide Solution

1a) binary operator+ is best implemented as a normal/friend function.
1b) unary operator- is best implemented as a member function.
1c) operator<< must be implemented as a normal/friend function.
1d) operator= must be implemented as a member function.

2) Write a class named Average that will keep track of the average of all integers passed to it. Use two members: The first one should be type int32_t, and used to keep track of the sum of all the numbers you've seen so far. The second should be of type int8_t, and used to keep track of how many numbers you've seen so far. You can divide them to find your average.

2a) Write all of the functions necessary for the following program to run:

```
int main()
{
    Average avg;

    avg += 4;
    std::cout << avg << '\n'; // 4 / 1 = 4

    avg += 8;
    std::cout << avg << '\n'; // (4 + 8) / 2 = 6

    avg += 24;
    std::cout << avg << '\n'; // (4 + 8 + 24) / 3 = 12

    avg += -10;
    std::cout << avg << '\n'; // (4 + 8 + 24 - 10) / 4 = 6.5

    (avg += 6) += 10; // 2 calls chained together
    std::cout << avg << '\n'; // (4 + 8 + 24 - 10 + 6 + 10) / 6 = 7

    Average copy = avg;
    std::cout << copy << '\n';

    return 0;
}
```

and produce the result:

```
4
6
12
6.5
7
7
```

Hint: Remember that int8_t is usually typedef'd as a char, so std::cout treats it accordingly.

Hide Solution

```
#include <iostream>
#include <cstdint> // for fixed width integers

class Average
{
private:
    int32_t m_total = 0; // the sum of all numbers we've seen so far
    int8_t m_numbers = 0; // the count of numbers we've seen so far

public:
    Average()
    {
    }

```

```
    friend std::ostream& operator<<(std::ostream &out, const Average &average)
    {
        // Our average is the sum of the numbers we've seen divided by the count of the numbers we've s
een
        // We need to remember to do a floating point division here, not an integer division
        out << static_cast<double>(average.m_total) / average.m_numbers;

        return out;
    }

    // Because operator+= modifies its left operand, we'll write it as a member
    Average& operator+=(int num)
    {
        // Increment our total by the new number
        m_total += num;
        // And increase the count by 1
        ++m_numbers;

        // return *this in case someone wants to chain +='s together
        return *this;
    }
};

int main()
{
    Average avg;

    avg += 4;
    std::cout << avg << '\n';

    avg += 8;
    std::cout << avg << '\n';

    avg += 24;
    std::cout << avg << '\n';

    avg += -10;
    std::cout << avg << '\n';

    (avg += 6) += 10; // 2 calls chained together
    std::cout << avg << '\n';

    Average copy = avg;
    std::cout << copy << '\n';

    return 0;
}
```

2b) Does this class need an explicit copy constructor or assignment operator?

Hide Solution

No. Because using memberwise initialization/copy is fine here, using the compiler provided defaults is acceptable.

3) Write your own integer array class named IntArray from scratch (do not use std::array or std::vector). Users should pass in the size of the array when it is created, and the array should be dynamically allocated. Use assert statements to guard against bad data. Create any constructors or overloaded operators needed to make the following program operate correctly:

```
#include <iostream>

IntArray fillArray()
{
    IntArray a(5);
    a[0] = 5;
    a[1] = 8;
    a[2] = 2;
    a[3] = 3;
```

```
    a[4] = 6;

    return a;
}

int main()
{
    IntArray a = fillArray();
    std::cout << a << '\n';

    IntArray b(1);
    a = a;
    b = a;

    std::cout << b << '\n';

    return 0;
}
```

This programs should print:

```
5 8 2 3 6
5 8 2 3 6
```

Hide Solution

```
#include <iostream>
#include <cassert> // for assert

class IntArray
{
private:
    int m_length = 0;
    int *m_array = nullptr;

public:
    IntArray(int length):
        m_length(length)
    {
        assert(length > 0 && "IntArray length should be a positive integer");

        m_array = new int[m_length] { 0 };
    }

    // Copy constructor that does a deep copy
    IntArray(const IntArray &array):
        m_length(array.m_length)
    {
        // Allocate a new array
        m_array = new int[m_length];

        // Copy elements from original array to new array
        for (int count = 0; count < array.m_length; ++count)
            m_array[count] = array.m_array[count];
    }

    ~IntArray()
    {
        delete[] m_array;
    }

    // If you're getting crazy values here you probably forgot to do a deep copy in your copy construc
tor
    friend std::ostream& operator<<(std::ostream &out, const IntArray &array)
    {
        for (int count = 0; count < array.m_length; ++count)
```

```
        {
            out << array.m_array[count] << ' ';
        }
        return out;
    }

    int& operator[] (const int index)
    {
        assert(index >= 0);
        assert(index < m_length);
        return m_array[index];
    }

    // Assignment operator that does a deep copy
    IntArray& operator= (const IntArray &array)
    {
        // self-assignment guard
        if (this == &array)
            return *this;

        // If this array already exists, delete it so we don't leak memory
        delete[] m_array;

        m_length = array.m_length;

        // Allocate a new array
        m_array = new int[m_length];

        // Copy elements from original array to new array
        for (int count = 0; count < array.m_length; ++count)
            m_array[count] = array.m_array[count];

        return *this;
    }

};

IntArray fillArray()
{
    IntArray a(5);
    a[0] = 5;
    a[1] = 8;
    a[2] = 2;
    a[3] = 3;
    a[4] = 6;

    return a;
}

int main()
{
    IntArray a = fillArray();

    // If you're getting crazy values here you probably forgot to do a deep copy in your copy construc
tor
    std::cout << a << '\n';

    IntArray b(1);
    a = a;
    b = a;

    // If you're getting crazy values here you probably forgot to do a deep copy in your assignment op
erator
    // or you forgot your self-assignment check
    std::cout << b << '\n';

    return 0;
```

```
}
```

4) Extra credit: This one is a little more tricky. A floating point number is a number with a decimal where the number of digits after the decimal can be variable. A fixed point number is a number with a fractional component where the number of digits in the fractional portion is fixed.

In this quiz, we're going to write a class to implement a fixed point number with two fractional digits (e.g. 12.34, 3.00, or 1278.99). Assume that the range of the class should be -32768.99 to 32767.99, that the fractional component should hold any two digits, that we don't want precision errors, and that we want to conserve space.

4a) What type of member variable(s) do you think we should use to implement our fixed point number with 2 digits after the decimal? (Make sure you read the answer before proceeding with the next questions)

Hide Solution

There are many different ways to implement a fixed point number. Because a fixed point number is essentially a subcase of a floating point number (where the number of digits after the decimal is fixed instead of variable), using a floating point number might seem like an obvious choice. But floating point numbers have precision issues. With a fixed number of decimal digits, we can reasonably enumerate all the possible fractional values (in our case, .00 to .99), so using a data type that has precision issues isn't the best choice.

A better solution would be to use a 16-bit signed integer to hold the non-fractional part of the number, and an 8-bit signed integer to hold the fractional component.

4b) Write a class named FixedPoint2 that implements the recommended solution from the previous question. If either (or both) of the non-fractional and fractional part of the number are negative, the number should be treated as negative. Provide the overloaded operators and constructors required for the following program to run:

```
int main()
{
    FixedPoint2 a(34, 56);
    std::cout << a << '\n';

    FixedPoint2 b(-2, 8);
    std::cout << b << '\n';

    FixedPoint2 c(2, -8);
    std::cout << c << '\n';

    FixedPoint2 d(-2, -8);
    std::cout << d << '\n';

    FixedPoint2 e(0, -5);
    std::cout << e << '\n';

    std::cout << static_cast<double>(e) << '\n';
```

```
    return 0;
}
```

This program should produce the result:

```
34.56
-2.08
-2.08
-2.08
-0.05
-0.05
```

Hint: Although it may initially seem like more work initially, it's helpful to store both the non-fractional and fractional parts of the number with the same sign (e.g. both positive if the number is positive, and both negative if the number is negative). This makes doing math much easier later.
Hint: To output your number, first cast it to a double.

Hide Solution

```
#include <iostream>
#include <cstdint> // for fixed width integers

class FixedPoint2
{
private:
    std::int16_t m_base; // here's our non-fractional part
    std::int8_t m_decimal; // here's our factional part

public:
    FixedPoint2(std::int16_t base = 0, std::int8_t decimal = 0)
        : m_base(base), m_decimal(decimal)
    {
        // We should handle the case where decimal is > 99 or < -99 here
        // but will leave as an exercise for the reader

        // If either the base or decimal or negative
        if (m_base < 0.0 || m_decimal < 0.0)
        {
            // Make sure base is negative
            if (m_base > 0.0)
                m_base = -m_base;
            // Make sure decimal is negative
            if (m_decimal > 0.0)
                m_decimal = -m_decimal;
        }
    }

    operator double() const
    {
        return m_base + static_cast<double>(m_decimal) / 100;
    }

    friend std::ostream& operator<<(std::ostream &out, const FixedPoint2 &fp)
    {
        out << static_cast<double>(fp);
        return out;
    }
};

int main()
{
    FixedPoint2 a(34, 56);
    std::cout << a << '\n';

```

```
    FixedPoint2 b(-2, 8);
    std::cout << b << '\n';

    FixedPoint2 c(2, -8);
    std::cout << c << '\n';

    FixedPoint2 d(-2, -8);
    std::cout << d << '\n';

    FixedPoint2 e(0, -5);
    std::cout << e << '\n';

    std::cout << static_cast<double>(e) << '\n';

    return 0;
}
```

4c) Now add a constructor that takes a double. You can round a number (on the left of the decimal) by using the round() function (included in header cmath).

Hint: You can get the non-fractional part of a double by static casting the double to an integer
Hint: To get the fractional part of a double, you'll first need to zero-out the non-fractional part. Use the integer value to do this.
Hint: You can move a digit from the right of the decimal to the left of the decimal by multiplying by 10. You can move it two digits by multiplying by 100.

The follow program should run:

```
int main()
{
    FixedPoint2 a(0.01);
    std::cout << a << '\n';

    FixedPoint2 b(-0.01);
    std::cout << b << '\n';

    FixedPoint2 c(5.01); // stored as 5.0099999... so we'll need to round this
    std::cout << c << '\n';

    FixedPoint2 d(-5.01); // stored as -5.0099999... so we'll need to round this
    std::cout << d << '\n';

    return 0;
}
```

This program should produce the result

```
0.01
-0.01
5.01
-5.01
```

Hide Solution

```
#include <iostream>
#include <cstdint> // for fixed width integers
#include <cmath> // for round()

class FixedPoint2
{
private:
    std::int16_t m_base; // here's our non-fractional part
    std::int8_t m_decimal; // here's our factional part

public:
    FixedPoint2(std::int16_t base = 0, std::int8_t decimal = 0)
```

```
        : m_base(base), m_decimal(decimal)
    {
        // We should handle the case where decimal is > 99 or < -99 here
        // but will leave as an exercise for the reader

        // If either the base or decimal or negative
        if (m_base < 0.0 || m_decimal < 0.0)
        {
            // Make sure base is negative
            if (m_base > 0.0)
                m_base = -m_base;
            // Make sure decimal is negative
            if (m_decimal > 0.0)
                m_decimal = -m_decimal;
        }
    }

    FixedPoint2(double d)
    {
        // First we need to get the non-fractional component
        // We can do this by casting our double to an integer
        m_base = static_cast<int16_t>(d); // truncates fraction

        // Now we need to get the fractional component:
        // 1) d - m_base leaves only the fractional portion
        // 2) which can we multiply by 100 to move the digits to the left of the decimal
        // 3) then we can round this
        // 4) and finally static cast to an integer to drop any extra decimals
        m_decimal = static_cast<std::int8_t>(round((d - m_base) * 100));
    }

    operator double() const
    {
        return m_base + static_cast<double>(m_decimal) / 100;
    }

    friend std::ostream& operator<<(std::ostream &out, const FixedPoint2 &fp)
    {
        out << static_cast<double>(fp);
        return out;
    }
};

int main()
{
    FixedPoint2 a(0.01);
    std::cout << a << '\n';

    FixedPoint2 b(-0.01);
    std::cout << b << '\n';

    FixedPoint2 c(5.01); // stored as 5.0099999... so we'll need to round this
    std::cout << c << '\n';

    FixedPoint2 d(-5.01); // stored as -5.0099999... so we'll need to round this
    std::cout << d << '\n';

    return 0;
}
```

4d) Overload operator==, operator >>, operator- (unary), and operator+ (binary).

The following program should run:

```
void testAddition()
{
    // h/t to reader Sharjeel Safdar for this function
```

```
    std::cout << std::boolalpha;
    std::cout << (FixedPoint2(0.75) + FixedPoint2(1.23) == FixedPoint2(1.98)) << '\n'; // both positiv
e, no decimal overflow
    std::cout << (FixedPoint2(0.75) + FixedPoint2(1.50) == FixedPoint2(2.25)) << '\n'; // both positiv
e, with decimal overflow
    std::cout << (FixedPoint2(-0.75) + FixedPoint2(-1.23) == FixedPoint2(-1.98)) << '\n'; // both negat
ive, no decimal overflow
    std::cout << (FixedPoint2(-0.75) + FixedPoint2(-1.50) == FixedPoint2(-2.25)) << '\n'; // both negat
ive, with decimal overflow
    std::cout << (FixedPoint2(0.75) + FixedPoint2(-1.23) == FixedPoint2(-0.48)) << '\n'; // second nega
tive, no decimal overflow
    std::cout << (FixedPoint2(0.75) + FixedPoint2(-1.50) == FixedPoint2(-0.75)) << '\n'; // second nega
tive, possible decimal overflow
    std::cout << (FixedPoint2(-0.75) + FixedPoint2(1.23) == FixedPoint2(0.48)) << '\n'; // first negati
ve, no decimal overflow
    std::cout << (FixedPoint2(-0.75) + FixedPoint2(1.50) == FixedPoint2(0.75)) << '\n'; // first negati
ve, possible decimal overflow
}

int main()
{
    testAddition();

    FixedPoint2 a(-0.48);
    std::cout << a << '\n';

    std::cout << -a << '\n';

    std::cout << "Enter a number: "; // enter 5.678
    std::cin >> a;

    std::cout << "You entered: " << a << '\n';

    return 0;
}
```

And produce the output:

```
true
true
true
true
true
true
true
true
-0.48
0.48
Enter a number: 5.678
You entered: 5.68
```

Hint: Add your two FixedPoint2 together by leveraging the double cast, adding the results, and converting back to a FixedPoint2.
Hint: For operator>>, use your double constructor to create an anonymous object of type FixedPoint2, and assign it to your FixedPoint2 function parameter

Hide Solution

```
#include <iostream>
#include <cstdint> // for fixed width integers
#include <cmath> // for round()

class FixedPoint2
{
private:
```

```
    std::int16_t m_base; // here's our non-fractional part
    std::int8_t m_decimal; // here's our factional part

public:
    FixedPoint2(std::int16_t base = 0, std::int8_t decimal = 0)
        : m_base(base), m_decimal(decimal)
    {
        // We should handle the case where decimal is > 99 or < -99 here
        // but will leave as an exercise for the reader

        // If either the base or decimal or negative
        if (m_base < 0.0 || m_decimal < 0.0)
        {
            // Make sure base is negative
            if (m_base > 0.0)
                m_base = -m_base;
            // Make sure decimal is negative
            if (m_decimal > 0.0)
                m_decimal = -m_decimal;
        }
    }

    FixedPoint2(double d)
    {
        // First we need to get the non-fractional component
        // We can do this by casting our double to an integer
        m_base = static_cast<int16_t>(d); // truncates fraction

        // Now we need to get the fractional component:
        // 1) d - m_base leaves only the fractional portion
        // 2) which can we multiply by 100 to move the digits to the left of the decimal
        // 3) then we can round this
        // 4) and finally static cast to an integer to drop any extra decimals
        m_decimal = static_cast<std::int8_t>(round((d - m_base) * 100));
    }

    operator double() const
    {
        return m_base + static_cast<double>(m_decimal) / 100;
    }

    friend bool operator==(const FixedPoint2 &fp1, const FixedPoint2 &fp2)
    {
        return (fp1.m_base == fp2.m_base && fp1.m_decimal == fp2.m_decimal);
    }

    friend std::ostream& operator<<(std::ostream &out, const FixedPoint2 &fp)
    {
        out << static_cast<double>(fp);
        return out;
    }

    friend std::istream& operator >> (std::istream &in, FixedPoint2 &fp)
    {
        double d;
        in >> d;
        fp = FixedPoint2(d);

        return in;
    }

    friend FixedPoint2 operator+(const FixedPoint2 &fp1, const FixedPoint2 &fp2)
    {
        return FixedPoint2(static_cast<double>(fp1) + static_cast<double>(fp2));
    }

    FixedPoint2 operator-()
```

```
    {
        return FixedPoint2(-m_base, -m_decimal);
    }

};

void testAddition()
{
        // h/t to reader Sharjeel Safdar for this method of testing
    std::cout << std::boolalpha;
    std::cout << (FixedPoint2(0.75) + FixedPoint2(1.23) == FixedPoint2(1.98)) << '\n'; // both positive, no decimal overflow
    std::cout << (FixedPoint2(0.75) + FixedPoint2(1.50) == FixedPoint2(2.25)) << '\n'; // both positive, with decimal overflow
    std::cout << (FixedPoint2(-0.75) + FixedPoint2(-1.23) == FixedPoint2(-1.98)) << '\n'; // both negative, no decimal overflow
    std::cout << (FixedPoint2(-0.75) + FixedPoint2(-1.50) == FixedPoint2(-2.25)) << '\n'; // both negative, with decimal overflow
    std::cout << (FixedPoint2(0.75) + FixedPoint2(-1.23) == FixedPoint2(-0.48)) << '\n'; // second negative, no decimal overflow
    std::cout << (FixedPoint2(0.75) + FixedPoint2(-1.50) == FixedPoint2(-0.75)) << '\n'; // second negative, possible decimal overflow
    std::cout << (FixedPoint2(-0.75) + FixedPoint2(1.23) == FixedPoint2(0.48)) << '\n'; // first negative, no decimal overflow
    std::cout << (FixedPoint2(-0.75) + FixedPoint2(1.50) == FixedPoint2(0.75)) << '\n'; // first negative, possible decimal overflow
}

int main()
{
    testAddition();

    FixedPoint2 a(-0.48);
    std::cout << a << '\n';

    std::cout << -a << '\n';

    std::cout << "Enter a number: "; // enter 5.678
    std::cin >> a;

    std::cout << "You entered: " << a << '\n';

    return 0;
}
```



10.1 -- Object relationships


Index


9.15 -- Shallow vs. deep copying