21.x — Chapter 21 summary and quiz

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 (unless it's handled naturally, or you're using the copy and swap idiom).

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

Question #1

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?

a) `point + point`

Show Solution

b) `-point`

Show Solution

c) `std::cout << point`

Show Solution

d) `point = 5;`

Show Solution

Question #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 `std::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 `std::int8_t` and used to keep track of how many numbers you've seen so far. You can divide them to find your average.

a) 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 `std::int8_t` is usually a `char` type, so `std::cout` treats them accordingly.

Show Solution

b) Should you define a copy constructor or assignment operator for this class?

Show Solution

Question #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';

	auto& ref{ a }; // we're using this reference to avoid compiler self-
assignment errors
	a = ref;

	IntArray b(1);
	b = a;

	a[4] = 7;

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

	return 0;
}
```

This program should print:

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

Show Solution

Question #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.

> Step #1

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)

Show Solution

> Step #2

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. For now, don't worry about the fractional portion being out of bounds (>99 or <-99).

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

int main()
{
        FixedPoint2 a{ 34, 56 };
        std::cout << a << '\n';
        std::cout << static_cast<double>(a) << '\n';
        assert(static_cast<double>(a) == 34.56);

        FixedPoint2 b{ -2, 8 };
        assert(static_cast<double>(b) == -2.08);

        FixedPoint2 c{ 2, -8 };
        assert(static_cast<double>(c) == -2.08);

        FixedPoint2 d{ -2, -8 };
        assert(static_cast<double>(d) == -2.08);

        FixedPoint2 e{ 0, -5 };
        assert(static_cast<double>(e) == -0.05);

        FixedPoint2 f{ 0, 10 };
        assert(static_cast<double>(f) == 0.1);

        return 0;
}
```

This program should produce the result:

```
34.56
34.56
```

Hint: To output your number, static_cast it to a double.

Show Solution

> Step #3

Now let’s handle the case where the fractional portion is out of bounds. We have two reasonable strategies here:

- Clamp the fraction portion (if >99, set to 99).
- Treat overflow as relevant (if >99, reduce by 99 and add 1 to base).

In this exercise, we’ll treat decimal overflow as relevant, as this will be useful in the next step.

The following should run:

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

// You will need to make testDecimal a friend of FixedPoint2
// so the function can access the private members of FixedPoint2
bool testDecimal(const FixedPoint2 &fp)
{
    if (fp.m_base >= 0)
        return fp.m_decimal >= 0 && fp.m_decimal < 100;
    else
        return fp.m_decimal <= 0 && fp.m_decimal > -100;
}

int main()
{
        FixedPoint2 a{ 1, 104 };
        std::cout << a << '\n';
        std::cout << static_cast<double>(a) << '\n';
        assert(static_cast<double>(a) == 2.04);
        assert(testDecimal(a));

        FixedPoint2 b{ 1, -104 };
        assert(static_cast<double>(b) == -2.04);
        assert(testDecimal(b));

        FixedPoint2 c{ -1, 104 };
        assert(static_cast<double>(c) == -2.04);
        assert(testDecimal(c));

        FixedPoint2 d{ -1, -104 };
        assert(static_cast<double>(d) == -2.04);
        assert(testDecimal(d));

        return 0;
}
```

And produce the output:

```
2.04
2.04
```

Show Solution

> Step #4

Now add a constructor that takes a double. The follow program should run:

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

int main()
{
        FixedPoint2 a{ 0.01 };
        assert(static_cast<double>(a) == 0.01);

        FixedPoint2 b{ -0.01 };
        assert(static_cast<double>(b) == -0.01);

        FixedPoint2 c{ 1.9 }; // make sure we handle single digit decimal
        assert(static_cast<double>(c) == 1.9);

        FixedPoint2 d{ 5.01 }; // stored as 5.0099999... so we'll need to round this
        assert(static_cast<double>(d) == 5.01);

        FixedPoint2 e{ -5.01 }; // stored as -5.0099999... so we'll need to round
this
        assert(static_cast<double>(e) == -5.01);

        // Handle case where the argument's decimal rounds to 100 (need to increase
base by 1)
        FixedPoint2 f { 106.9978 }; // should be stored with base 107 and decimal 0
        assert(static_cast<double>(f) == 107.0);

        // Handle case where the argument's decimal rounds to -100 (need to decrease
base by 1)
        FixedPoint2 g { -106.9978 }; // should be stored with base -107 and decimal 0
        assert(static_cast<double>(g) == -107.0);

        return 0;
}
```

Recommendation: This one will be a bit tricky. Do this one in three steps. First, solve for the cases where the double parameter is representable directly (variables `a` through `c` above). Then, update your code to handle the cases where the double parameter has a rounding error (variables `d` & `e`). Variables `f` and `g` should be handled by the overflow handling we added in the prior step.

For all cases: Show Hint

For variables `a` through `c`: Show Hint

For variables `d` & `e`: Show Hint

Show Solution

> Step #5

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

The following program should run:

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

int main()
{
	assert(FixedPoint2{ 0.75 } == FixedPoint2{ 0.75 });    // Test equality true
	assert(!(FixedPoint2{ 0.75 } == FixedPoint2{ 0.76 })); // Test equality false

	// Test additional cases -- h/t to reader Sharjeel Safdar for these test
cases
	assert(FixedPoint2{ 0.75 } + FixedPoint2{ 1.23 } == FixedPoint2{ 1.98 });
// both positive, no decimal overflow
	assert(FixedPoint2{ 0.75 } + FixedPoint2{ 1.50 } == FixedPoint2{ 2.25 });
// both positive, with decimal overflow
	assert(FixedPoint2{ -0.75 } + FixedPoint2{ -1.23 } == FixedPoint2{ -1.98 });
// both negative, no decimal overflow
	assert(FixedPoint2{ -0.75 } + FixedPoint2{ -1.50 } == FixedPoint2{ -2.25 });
// both negative, with decimal overflow
	assert(FixedPoint2{ 0.75 } + FixedPoint2{ -1.23 } == FixedPoint2{ -0.48 });
// second negative, no decimal overflow
	assert(FixedPoint2{ 0.75 } + FixedPoint2{ -1.50 } == FixedPoint2{ -0.75 });
// second negative, possible decimal overflow
	assert(FixedPoint2{ -0.75 } + FixedPoint2{ 1.23 } == FixedPoint2{ 0.48 });
// first negative, no decimal overflow
	assert(FixedPoint2{ -0.75 } + FixedPoint2{ 1.50 } == FixedPoint2{ 0.75 });
// first negative, possible decimal overflow

	FixedPoint2 a{ -0.48 };
	assert(static_cast<double>(a) == -0.48);
	assert(static_cast<double>(-a) == 0.48);

	std::cout << "Enter a number: "; // enter 5.678
	std::cin >> a;
	std::cout << "You entered: " << a << '\n';
	assert(static_cast<double>(a) == 5.68);

	return 0;
}
```

Show Hint

Show Hint

Show Solution