13.x — Chapter 13 comprehensive quiz

ALEX AUGUST 20, 2021

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

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;

Show Solution

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::int_least32_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::int_least8_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 8 bit integers are usually `char`s, so `std::cout` treats them accordingly.

Show Solution

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

Show Solution

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;

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

 return 0;
}
```

This program should print:

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

Show Solution

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)

Show Solution

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: To output your number, first cast it to a double.

Show Solution

4c) Now add a constructor that takes a double. The follow program should run:

```
int main()
{
 // Handle cases where the argument is representable directly
 FixedPoint2 a{ 0.01 };
 std::cout << a << '\n';

 FixedPoint2 b{ -0.01 };
 std::cout << b << '\n';

 // Handle cases where the argument has some rounding error
 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';

 // Handle case where the argument's decimal rounds to 100 (need to increase base by 1)
 FixedPoint2 e{ 106.9978 }; // should be stored with base 107 and decimal 0
 std::cout << e << '\n';

 return 0;
}
```

This program should produce the result

```
0.01
-0.01
5.01
-5.01
107
```

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 (cases a & b above). Then, update your code to handle the cases where the double parameter has a rounding

error (cases c & d). Lastly, handle the edge case where the decimal rounds up to 100 (case e).

For all cases: Show Hint

For cases a & b: Show Hint

For cases c & d: Show Hint

Show Solution

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 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;
}
```

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

Show Solution

Next lesson
16.1 Object relationships

Back to table of contents

Previous lesson
13.17 Overloading operators and function templates

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ co

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest