9.7 — Overloading the increment and decrement operators

Overloading the increment (++) and decrement (--) operators are pretty straightforward, with one small exception. There are actually two versions of the increment and decrement operators: a prefix increment and decrement (e.g. ++x; --y;) and a postfix increment and decrement (e.g. x++; y--;).

Because the increment and decrement operators are both unary operators and they modify their operands, they're best overloaded as member functions. We'll tackle the prefix versions first because they're the most straightforward.

Overloading prefix increment and decrement

Prefix increment and decrement is overloaded exactly the same as any normal unary operator. We'll do this one by example:

```cpp
#include <iostream>

class Digit
{
private:
    int m_digit;
public:
    Digit(int digit=0)
        : m_digit(digit)
    {
    }

    Digit& operator++();
    Digit& operator--();

    friend std::ostream& operator<< (std::ostream &out, const Digit &d);
};

Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_digit == 9)
        m_digit = 0;
    // otherwise just increment to next number
    else
        ++m_digit;

    return *this;
}

Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_digit == 0)
        m_digit = 9;
    // otherwise just decrement to next number
    else
        --m_digit;

    return *this;
}

std::ostream& operator<< (std::ostream &out, const Digit &d)
{
    out << d.m_digit;
    return out;
}

int main()
```

```
50   {
51       Digit digit(8);
52
53       std::cout << digit;
54       std::cout << ++digit;
55       std::cout << ++digit;
56       std::cout << --digit;
57       std::cout << --digit;
58
59       return 0;
60   }
```

Our Digit class holds a number between 0 and 9. We've overloaded increment and decrement so they increment/decrement the digit, wrapping around if the digit is incremented/decremented out range.

This example prints:

89098

Note that we return *this. The overloaded increment and decrement operators return the current implicit object so multiple operators can be "chained" together.

Overloading postfix increment and decrement

Normally, functions can be overloaded when they have the same name but a different number and/or different type of parameters. However, consider the case of the prefix and postfix increment and decrement operators. Both have the same name (eg. operator++), are unary, and take one parameter of the same type. So how it is possible to differentiate the two when overloading?

The answer is that C++ uses a "dummy variable" or "dummy argument" for the postfix operators. This argument is a fake integer parameter that only serves to distinguish the postfix version of increment/decrement from the prefix version. Here is the above Digit class with both prefix and postfix overloads:

```
1    class Digit
2    {
3    private:
4        int m_digit;
5    public:
6        Digit(int digit=0)
7            : m_digit(digit)
8        {
9        }
10
11       Digit& operator++(); // prefix
12       Digit& operator--(); // prefix
13
14       Digit operator++(int); // postfix
15       Digit operator--(int); // postfix
16
17       friend std::ostream& operator<< (std::ostream &out, const Digit &d);
18   };
19
20   Digit& Digit::operator++()
21   {
22       // If our number is already at 9, wrap around to 0
23       if (m_digit == 9)
24           m_digit = 0;
25       // otherwise just increment to next number
26       else
27           ++m_digit;
28
29       return *this;
30   }
31
32   Digit& Digit::operator--()
33   {
```

```cpp
34            // If our number is already at 0, wrap around to 9
35            if (m_digit == 0)
36                m_digit = 9;
37            // otherwise just decrement to next number
38            else
39                --m_digit;
40
41            return *this;
42        }
43
44    Digit Digit::operator++(int)
45        {
46            // Create a temporary variable with our current digit
47            Digit temp(m_digit);
48
49            // Use prefix operator to increment this digit
50            ++(*this); // apply operator
51
52            // return temporary result
53            return temp; // return saved state
54        }
55
56    Digit Digit::operator--(int)
57        {
58            // Create a temporary variable with our current digit
59            Digit temp(m_digit);
60
61            // Use prefix operator to decrement this digit
62            --(*this); // apply operator
63
64            // return temporary result
65            return temp; // return saved state
66        }
67
68    std::ostream& operator<< (std::ostream &out, const Digit &d)
69        {
70            out << d.m_digit;
71            return out;
72        }
73
74    int main()
75        {
76            Digit digit(5);
77
78            std::cout << digit;
79            std::cout << ++digit; // calls Digit::operator++();
80            std::cout << digit++; // calls Digit::operator++(int);
81            std::cout << digit;
82            std::cout << --digit; // calls Digit::operator--();
83            std::cout << digit--; // calls Digit::operator--(int);
84            std::cout << digit;
85
86            return 0;
87        }
```

This prints

5667665

There are a few interesting things going on here. First, note that we've distinguished the prefix from the postfix operators by providing an integer dummy parameter on the postfix version. Second, because the dummy parameter is not used in the function implementation, we have not even given it a name. This tells the compiler to treat this variable as a placeholder, which means it won't warn us that we declared a variable but never used it.

Third, note that the prefix and postfix operators do the same job -- they both increment or decrement the object. The difference between the two is in the value they return. The overloaded prefix operators return the object after it has been incremented or decremented. Consequently, overloading these is fairly straightforward. We simply increment or decrement our member variables, and then return *this.

The postfix operators, on the other hand, need to return the state of the object *before* it is incremented or decremented. This leads to a bit of a conundrum -- if we increment or decrement the object, we won't be able to return the state of the object before it was incremented or decremented. On the other hand, if we return the state of the object before we increment or decrement it, the increment or decrement will never be called.

The typical way this problem is solved is to use a temporary variable that holds the value of the object before it is incremented or decremented. Then the object itself can be incremented or decremented. And finally, the temporary variable is returned to the caller. In this way, the caller receives a copy of the object before it was incremented or decremented, but the object itself is incremented or decremented. Note that this means the return value of the overloaded operator must be a non-reference, because we can't return a reference to a local variable that will be destroyed when the function exits. Also note that this means the postfix operators are typically less efficient than the prefix operators because of the added overhead of instantiating a temporary variable and returning by value instead of reference.

Finally, note that we've written the post-increment and post-decrement in such a way that it calls the pre-increment and pre-decrement to do most of the work. This cuts down on duplicate code, and makes our class easier to modify in the future.

Share this:

 Facebook   Twitter   G+ Google   Pinterest

92 comments to 9.7 — Overloading the increment and decrement operators