13.8 — Overloading the increment and decrement operators

ALEX JULY 31, 2021

Overloading the increment (`++`) and decrement (`--`) operators is 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 are overloaded exactly the same as any normal unary operator. We'll do this one by example:

```
#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()
{
    Digit digit(8);

    std::cout << digit;
    std::cout << ++digit;
    std::cout << ++digit;
    std::cout << --digit;
    std::cout << --digit;

    return 0;
}
```

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 C++ language specification has a special case that provides the answer: the compiler looks to see if the overloaded operator has an int parameter. If the overloaded operator has an int parameter, the operator is a postfix overload. If the overloaded operator has no parameter, the operator is a prefix overload.

Here is the above Digit class with both prefix and postfix overloads:

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

    Digit& operator++(); // prefix has no parameter
    Digit& operator--(); // prefix has no parameter

    Digit operator++(int); // postfix has an int parameter
    Digit operator--(int); // postfix has an int parameter

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

// No parameter means this is prefix operator++
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;
}

// No parameter means this is prefix operator--
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;
}

// int parameter means this is postfix operator++
Digit Digit::operator++(int)
{
    // Create a temporary variable with our current digit
    Digit temp{*this};

    // Use prefix operator to increment this digit
    ++(*this); // apply operator

    // return temporary result
    return temp; // return saved state
}

// int parameter means this is postfix operator--
Digit Digit::operator--(int)
{
    // Create a temporary variable with our current digit
    Digit temp{*this};

    // Use prefix operator to decrement this digit
    --(*this); // apply operator

    // return temporary result
    return temp; // return saved state
}

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

int main()
{
    Digit digit(5);

    std::cout << digit;
    std::cout << ++digit; // calls Digit::operator++();
    std::cout << digit++; // calls Digit::operator++(int);
    std::cout << digit;
    std::cout << --digit; // calls Digit::operator--();
    std::cout << digit--; // calls Digit::operator--(int);
    std::cout << digit;

    return 0;
}
```

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.

Next lesson
13.9 Overloading the subscript operator

Back to table of contents

Previous lesson
13.7 Overloading the comparison operators

B | U | URL | INLINE CODE | C++ CODE BLOCK | HELP!

Leave a comment...

Name*

Email*

Notify me about replies: POST COMMENT

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

157 COMMENTS Newest

Mateusz Kacpersi
January 27, 2022 9:30 am

"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. "

I get it ;D sory...

0 Reply

Mateusz Kacpersi
January 27, 2022 9:22 am

Hello :)

Why we didn't use reference '&' next to Digit data type ? ?

```
// int parameter means this is postfix operator++
Digit Digit::operator++(int)
{
    // Create a temporary variable with our current digit
    Digit temp{*this};

    // Use prefix operator to increment this digit
    ++(*this); // apply operator

    // return temporary result
    return temp; // return saved state
}
```

Thx <3

0 Reply

> **Alex** Author
> Reply to Mateusz Kacpersi January 27, 2022 8:06 pm
>
> I'm going to presume you're asking why we didn't return a `Digit&` instead of a `Digit`. We're returning a local variable that will get destroyed at the end of the function. If we return a reference to a local variable, the variable will be destroyed before the caller has access to it, and the returned pointer will be dangling.
>
> 1 Reply

CHIMMILI VINAYKUMAR
November 19, 2021 10:43 pm

hey Alex i have a small confession for you, please reduce the add content on-page, don't get me wrong I know this is required for you to maintain page expenses and all, but the page really looks messy with all this add content, why don't you try something like a short video add for a page or two like youtube if possible and keep the page add free without adds.

0 Reply

> **Michael Butler**
> Reply to CHIMMILI VINAYKUMAR November 24, 2021 3:14 am
>
> You should consider adding a donation / buy me a coffee section to the footer on this note, I would have happily done so with all of this great content provided for free
>
> 3 Reply

Tejero Joshua
August 18, 2021 10:31 pm

even though I choose to return the object by value but it's still applied the change the overload operator made, is there other reason why we return by reference except it doesn't create redundant copy of the object?

0 Reply

> **Alex** Author
> Reply to Tejero Joshua August 19, 2021 9:58 am
>
> We return by reference so you can chain multiple operators together in the same expression. Otherwise the second evaluated operator would be applied to the returned copy, not the original object.
>
> 0 Reply

Waldo Lemmer
July 21, 2021 2:14 am

- Everywhere: `type &identifier`
- First sentence:

> Overloading the increment (++) and decrement (--) operators are pretty straightforward

"are" should be "is" since "overloading" is a singular gerund (https://portlandenglish.edu/blog/gerund-subjects/)

- Section "Overloading prefix increment and decrement":

> Prefix increment and decrement is overloaded exactly the same as any normal unary operator

"is" should be "are" (https://editorsmanual.com/articles/compound-subject-singular-or-plural/#compound-subject-containing-and)

These seemingly arbitrary English grammar rules are so confusing lol

0 Reply

Husen Patel
July 12, 2021 10:12 pm

Wouldn't it work well?

```
Digit Digit::operator++(int)
{
    return m_digit++;
    //or
    return ++m_digit;
    //or
        return ++(*this);
}
```

1 Reply

> **Waldo Lemmer**
> Reply to Husen Patel July 21, 2021 2:22 am
>
> 1 and 2 won't work if new members are added (unlike the one in the lesson)
> 2 and 3 behave the same as Digit::operator++() because they don't return the temporary copies
>
> 0 Reply

Terry
June 24, 2021 11:51 pm

I've looked at C/C++ and similar languages like Java many times before and have never warmed to the idea/use of these operators preferring to explicitly code

```
x = x + 1;
```

as opposed to

```
x++;
```

with the first being easy to read and decipher what's going on at a glance. However you've just presented a very good reason why I need to rethink that stance with operator overloading. The idea of automatically and safely rotating an index is powerful

```
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;
}
```

0 Reply

> **beaners**
> Reply to Terry July 29, 2021 1:35 am
>
> I think you meant ++x; instead of x++;
>
> Pre-fix increment operators do what your code do on the first line however you used a post-fix operator.
>
> 0 Reply

Alireza Nikpay
April 5, 2021 12:24 am

Nice explanation

```
Digit temp(m_digit);
--(*this);
return temp;
```

can be written like

```
return Digit{ m_digit++ };
```

right?

0 Reply

> **nascardriver** Sub-admin
> Reply to Alireza Nikpay April 6, 2021 7:20 am
>
> Smart, yes that works!
>
> I've updated the example to construct `temp` from `*this`, making it more portable.
>
> 0 Reply

J34NP3T3R
February 5, 2021 7:57 pm

we could use anonymous return ?

```
Digit Digit::operator++(int)
{
    // Create a temporary variable with our current digit
    //Digit temp(m_digit++);

    // Use prefix operator to increment this digit
    //++(*this); // apply operator

    // return temporary result
    return { m_digit++ }; // return saved state then increment
}
```

0 Reply

> **JustAPleb**
> Reply to J34NP3T3R March 2, 2021 5:26 pm
>
> your code is the equivalent of an infinite recursive function.the function has been overloaded but you are calling the same function over and over again.
>
> ```
> return {m_digit++};  // will cause the execution to go to Digit Digit::operator++(int)
> ```
>
> 0 Reply

>> **J34NP3T3R**
>> Reply to JustAPleb March 2, 2021 5:34 pm
>>
>> really ? ... i thought it wasn't...
>>
>> m_digit++ is not an overload of Digit::operator++
>>
>> m_digit is an integer ...
>>
>> 1 Reply

SaMIRa
August 24, 2020 9:47 am

Hello,

Is the following a bad-written code?I got the same result as yours.

```
Digit Digit::operator++(int)
{
     if (m_digit == 10)
         m_digit = 0;
     return { this->m_digit++ };
}

Digit Digit::operator--(int)
{
    if (m_digit == -1)
        m_digit = 9;
    return {this->m_digit--};
}
```

>>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.
Did you mean my 'return' code part in the code above?

>>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.
Is this how such operators are implemented in the C++ language itself?Now it makes sense why you asked us to use ++x instead of x++ :)

0 Reply

> **nascardriver** Sub-admin
> Reply to SaMIRa August 25, 2020 5:45 am
>
> If `operator++` only increases `m_digit`, your solution is fine too. If `operator++` did a lot of other things as well, you'd have duplicate code in the ++prefix and ++postfix overloads. By calling ++prefix from postfix++, we're guaranteed that they behave the same (Only the return value is different).
>
> I'm just as confused as you about the sentence about incrementing after returning. Maybe Alex means
>
> ```
> {
>   return *this;
>   ++this->m_digit;
> }
> ```
>
> Fundamental types (`int`, etc.) don't have implementations. The compiler just knows how operations on these types work. But yes, ++ and -- work the same for these types. If you use postfix++, the variable has to be copied. However, for fundamental types, the compiler will most likely change postfix++ to ++prefix if it notices that you don't use the return value of postfix++.
>
> 0 Reply

>> **SaMIRa**
>> Reply to nascardriver August 25, 2020 12:25 pm
>>
>> Thank you so much dear nascardriver. <3 <3
>>
>> 0 Reply

©2022 Learn C++