9.7 — Overloading the increment and decrement operators

BY ALEX ON OCTOBER 15TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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:

```
#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;
}

```

```cpp
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 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:

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

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

    Digit operator++(int); // postfix
    Digit operator--(int); // postfix

    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;
}

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 temp; // return saved state
}

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

    // 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.

9.8 -- Overloading the subscript operator

Index

9.6 -- Overloading the comparison operators

C++ TUTORIAL | PRINT THIS POST

127 comments to 9.7 — Overloading the increment and decrement operators

« Older Comments 1 2

Ged
January 21, 2020 at 9:50 am · Reply

I have a quick question.

```
Digit Digit::operator++(int)
```

How does it understand that we want - example++ instead of ++example?

situation1 - ++(int)
situation2 - (int)++

By adding a dummy argument int to the function our both functions become different, but how does the compiler know which is which?

nascardriver
January 22, 2020 at 3:52 am · Reply

This is a special rule for the ++ operator. It's not something that can be done manually.

Umair Malik
January 2, 2020 at 5:33 am · Reply

I have a query about postfix increment/decrement operator.
Kindly tell why we write it as

Digit &Digit::operator++(int)
instea of
Digit &Digit::operator++( )
mean why we use 'int'?
kindly explain.

nascardriver
January 2, 2020 at 5:43 am · Reply

It's been defined like that. The overload without parameters is the prefix operater, the overload with an argument is the postfix operator. There is no argument to this function. I guess this was the easiest way of differentiating them without making changes to the language.

Umair
January 2, 2020 at 11:42 pm · Reply

ok
Thanks for your response

Shekhar
December 25, 2019 at 3:34 am · Reply

Hi Alex ,

Just wanted to clear a doubt .I have tried to overload the pre-increment operator in two version.I am not sure what is wrong with the Second version.
Kindly throw some light over this.

I have defined a Digit class having two datamembers a and b
Approach-1:

```
#include<iostream>
using namespace std;

class Digit
{
    int a,b;
    public:
        Digit(int a=0, int b=0) : a(a), b(b){}
        Digit& operator++()
        {
            ++(*this);
            return *this;


        }
        Digit& operator--()
        {

            --a;
            --b;
            return *this;
        }

        friend ostream& operator<<(ostream& out , const Digit& d)
        {
            out << " ( " << d.a << " , " << d.b << " )\n";
            return out;
        }
};

int main()
{
    Digit d(10,20);
    cout << d;
    cout << ++d;
    return 0;
}
```

Version 2:

```
Digit& operator++()
       {
           a++;
           b++;
           return *this;


       }
```

Approach 2 is working fine but Approah 1 is not working while overloading pre-increment ++ operator

nascardriver
December 27, 2019 at 7:42 am · Reply

Version 1 is recursively calling itself. That's an infinite loop.

Chandra
December 25, 2019 at 2:32 am · Reply

Working Code ::Chandra Shekhar

```
#include<iostream>
using namespace std;

class Digit
{
    int num,bun;
    public:
    Digit(int num = 0,int bun=0):num(num),bun(bun){}

    Digit& operator++();
    Digit& operator--();
    Digit operator++(int);
    friend ostream &operator<<(ostream &out, const Digit &d);
};

Digit& Digit::operator++()
{
    ++num;
    ++bun;
    return *this;
}

Digit& Digit::operator--()
{
    --num;
    --bun;
    return *this;
}

//post increment
Digit Digit::operator++(int)
{
    Digit temp;
    temp.num = num;
    temp.bun = bun;
    ++(*this);
    return temp;
}

ostream &operator<<(ostream &out, const Digit &d)
{
    out << " ( " << d.num << " , " << d.bun << "  ) \n";
    return out;
}

int main()
{
    Digit d1(10,11);
    cout << "d1  original state: " << d1 << endl;
    cout << "Incremeting using ++ pre opeartor \n";
    ++d1;
    cout << "After applying the pre++ operator : " << d1 << endl;

    cout << "Post increment operator : \n";
    cout << d1++;
}
```

Vishs
October 30, 2019 at 10:58 pm · Reply

Hi,

For the postfix overloading, you have done:

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

Why cant we simply do the below:

```
Digit Digit::operator++(int)
{
    return (m_digit++); // return saved state
}
```

This produces the exact same output!

nascardriver
October 31, 2019 at 4:17 am · Reply

Hi!

Your suggestion works for the `Digit` class, so yes, we could do it. As soon as there's more than one member in the class, your suggestion no longer works and would have to be changed to what is shown in the lesson.

Eru
August 23, 2019 at 2:05 pm · Reply

Hello, I've asked a question in chapter 9.3 about chaining operators earlier, so I'm still trying to wrap my head around them.

I'm guessing that the following code

```
SomeClass val;
----val;
```

would be considered as chaining, and this works because we're returning the val by reference, so each returned val after the operator-- function is the same val we've started with.(if I'm correct with my assumption here)

Now by this logic, the post-fix increment or decrement should not work for chaining(val----) and if I haven't done any mistakes while testing, it indeed does not work. (Because the value returned from the operator--(int x) is not the val we've started with, it's just a copy with a different address)

Also by doesn't work i mean it still gets evaluated correctly to the value, but it(the chained operators) does not affect the original value, as in:

```
    Digit digit(5);

    std::cout << digit++++++; // calls Digit::operator++(int);
    std::cout << digit;
    std::cout << digit--; // calls Digit::operator--(int);
    std::cout << digit;
    // This prints out 5 6 6 5, instead of 5 8 8 7.
```

However the pre-fix version works correctly, as in;

```
Digit digit(5);


    std::cout << ++++++digit; // calls Digit::operator++(int);
    std::cout << digit;
    std::cout << --digit; // calls Digit::operator--(int);
    std::cout << digit;
    //this prints 8 8 7 7, changing the digit we've defined
```

I would be glad if anyone can answer whether what I wrote is correct or not, the whole thing is too complicated! :D

nascardriver
August 24, 2019 at 2:06 am · Reply

Hello,

I didn't reply to your previous comment, because I don't know the definition of "chaining", or if there even is a proper definition.
I'd say chaining is whenever you apply a function immediately to the call of another function.
Your tests and explanation for the results are absolutely right.

Eru
August 24, 2019 at 4:33 am · Reply

Hey, thank you so much for replying, it feels like getting c++ correct is a really difficult task.

Thanks again :)

Alex
August 24, 2019 at 1:31 pm · Reply

https://en.wikipedia.org/wiki/Method_chaining

Eru
August 25, 2019 at 4:18 am · Reply

Hey, the link you provided is great! Should have done a little more research before posting :D

Thanks for replying :)

Paulo Filipe
July 13, 2019 at 3:10 am · Reply

Guys, here's a situation where a simple mistake of using prefix vs postfix caused a major memory leak in CloudFlare. Loads of private data exposed because of a simple

```
++p
```

instead of

```
p++
```

https://www.google.com/amp/s/blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/amp/

Atas
July 8, 2019 at 1:39 am · Reply

How does the syntax work out when we call the postfix version of ++? Where does the dummy variable come from to tell the compiler we want the postfix ++? How does compiler know to look to the left of operator++ to find the object, if all other unary operators have the object to the right? Thanks!

Alex
July 9, 2019 at 10:32 am · Reply

Short answer: The C++ specification specifies that it should be so, and it's up to the compiler implementers to figure this out.

lyf
April 17, 2019 at 1:53 am · Reply

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

    // Use prefix operator to increment this digit
```

```
 7      ++(*this); // apply operator
 8
 9      // return temporary result
10      return temp; // return saved state
11  }
```

is temp will be destroyed after function exits? maybe should use new Digit ?

nascardriver
April 17, 2019 at 3:07 am · Reply

@temp is returned by copy.

hassan magaji
March 5, 2019 at 12:56 pm · Reply

Hi,
i took @Alex's code and modified it a bit, instead of instantiating two objects of Digit, its now one:

```
 1  Digit Digit::operator++(int)
 2  {
 3      // Create a temporary(int) variable to hold our current this->m_digit
 4      int temp{ m_digit };
 5
 6      // Use prefix operator to increment this digit
 7      ++(*this); // apply operator
 8
 9      // return temporary result initialized with the prevoius value of this->m_digit stored
10      return { temp }; // return saved state
11  }
```

could this enhance performance?

nascardriver
March 6, 2019 at 5:22 am · Reply

Line 10 creates a new instance.

hassan magaji
March 6, 2019 at 5:48 am · Reply

yes @Alex line 47 and 53(copied into the return value) creates two instances right?
I think making copies/instances should be minimized so i took that approach, is my solution better/worse/thesame?

thank you alot nas.... looking forward for your reply.

nascardriver
March 6, 2019 at 5:53 am · Reply

Oops, I didn't see that.
Yes, your approach could be faster!
In reality, the compiler most likely optimizes yours and Alex' code such that they turn out the same. If it doesn't, your code is faster.

hassan magaji
March 7, 2019 at 4:55 am · Reply

thanks you nas...
by the way, do you know a site where i can get C++ updates (cpp11-20)?

nascardriver
March 7, 2019 at 8:17 am · Reply

You don't install cpp standard updates, you install a compiler that supports a new standard.
Install the latest version of your compiler and you should be good to go.

hassan magaji
March 7, 2019 at 2:20 pm · Reply

thanks

nascardriver
February 7, 2019 at 6:34 am · Reply

You have

```
Digit &Digit::operator++(int)
// or
Digit &Digit::operator--(int)
```

Remove the '&'. If your code causes errors, please post your code. An error message alone is rarely informative enough to help.

Chandra
February 7, 2019 at 4:30 am · Reply

Just had a query :

For overloading Post incr/decr operator here we used Digit operator++(int) but for pre increment operator overloading we have used Digit& operator().

My question is why the return types are different ie Digit and Digit&.

Kindly help with the explanation.

Kind Regards,
A C++ Seeker :)

nascardriver
February 7, 2019 at 6:35 am · Reply

Postfix++ has to create a copy of the object, as it returns the old value.
++Prefix returns the value after incrementing it, so it can just return the object itself.

Chandra

February 7, 2019 at 4:10 am · Reply

While Compiling Giving Warning:
|warning: reference to local variable 'temp' returned [-Wreturn-local-addr]

Also in the output screen not able to get anything

« Older Comments 1 2