9.5 — Overloading unary operators +, -, and !

BY ALEX ON OCTOBER 8TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Overloading unary operators

Unlike the operators you've seen so far, the positive (+), negative (-) and logical not (!) operators all are unary operators, which means they only operate on one operand. Because they only operate on the object they are applied to, typically unary operator overloads are implemented as member functions. All three operands are implemented in an identical manner.

Let's take a look at how we'd implement operator- on the Cents class we used in a previous example:

```
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    // Overload -Cents as a member function
    Cents operator-() const;

    int getCents() const { return m_cents; }
};

// note: this function is a member function!
Cents Cents::operator-() const
{
    return Cents(-m_cents);
}

int main()
{
    const Cents nickle(5);
    std::cout << "A nickle of debt is worth " << (-nickle).getCents() << " cents\n";

    return 0;
}
```

This should be straightforward. Our overloaded negative operator (-) is a unary operator implemented as a member function, so it takes no parameters (it operates on the *this object). It returns a Cents object that is the negation of the original Cents value. Because operator- does not modify the Cents object, we can (and should) make it a const function (so it can be called on const Cents objects).

Note that there's no confusion between the negative operator- and the minus operator- since they have a different number of parameters.

Here's another example. The ! operator is the logical negation operator -- if an expression evaluates to "true", operator! will return false, and vice-versa. We commonly see this applied to boolean variables to test whether they are true or not:

```
if (!isHappy)
    std::cout << "I am not happy!\n";
else
    std::cout << "I am so happy!\n";
```

For integers, 0 evaluates to false, and anything else to true, so operator! as applied to integers will return true for an integer value of 0 and false otherwise.

Extending the concept, we can say that operator! should evaluate to true if the state of the object is “false”, “zero”, or whatever the default initialization state is.

The following example shows an overload of both operator- and operator! for a user-defined Point class:

```
#include <iostream>

class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0):
        m_x(x), m_y(y), m_z(z)
    {
    }

    // Convert a Point into its negative equivalent
    Point operator- () const;

    // Return true if the point is set at the origin
    bool operator! () const;

    double getX() const { return m_x; }
    double getY() const { return m_y; }
    double getZ() const { return m_z; }
};

// Convert a Point into its negative equivalent
Point Point::operator- () const
{
    return Point(-m_x, -m_y, -m_z);
}

// Return true if the point is set at the origin, false otherwise
bool Point::operator! () const
{
    return (m_x == 0.0 && m_y == 0.0 && m_z == 0.0);
}

int main()
{
    Point point; // use default constructor to set to (0.0, 0.0, 0.0)

    if (!point)
        std::cout << "point is set at the origin.\n";
    else
        std::cout << "point is not set at the origin.\n";

    return 0;
}
```

The overloaded operator! for this class returns the boolean value “true” if the Point is set to the default value at coordinate (0.0, 0.0, 0.0). Thus, the above code produces the result:

```
point is set at the origin.
```

Quiz time

1) Implement overloaded operator+ for the Point class.

Show Solution

9.6 -- Overloading the comparison operators

Index

9.4 -- Overloading operators using member functions

C++ TUTORIAL | PRINT THIS POST

91 comments to 9.5 — Overloading unary operators +, -, and !

« Older Comments 1 2

David
January 18, 2020 at 5:07 am · Reply

Hello,I have a question in your example. Before we call the operator- , the value of nickle is 5. However,when we call the operator-(),the value of nickle becomes -5. And why can you say the operator-() doesn't modify the object?
In my opinion , its behavior violated the const meaning.
Does my concept be wrong?

```
1  #include <iostream>
2
3  class Cents
4  {
5  private:
```

```
6       int m_cents;
7
8   public:
9       Cents(int cents) { m_cents = cents; }
10
11      // Overload -Cents as a member function
12      Cents operator-() const;
13
14      int getCents() const { return m_cents; }
15  };
16
17  // note: this function is a member function!
18  Cents Cents::operator-() const
19  {
20      return Cents(-m_cents);
21  }
22
23  int main()
24  {
25      const Cents nickle(5);
26      std::cout << "A nickle of debt is worth " << (-nickle).getCents() << " cents\n";
27
28      return 0;
29  }
```

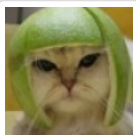
nascardriver
January 18, 2020 at 5:10 am · Reply

The value of `nickle` doesn't change. `operator-` creates a new `Cents` object with value -5. You can print `nickle.getCents()` in line 27 to verify this.

Ged
January 6, 2020 at 2:38 pm · Reply

Is there any difference if this function is outside of the class or inside?

```
1   Cents Cents::operator-() const
2   {
3       return Cents(-m_cents);
4   }
```



Alex
January 8, 2020 at 11:50 am · Reply

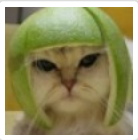
Functions defined inside the class are assumed to be inline. Other than that, I'm not aware of any differences.

hellmet
November 16, 2019 at 5:30 am · Reply

I was wondering why not have the operator- () function be left in the class itself?

Alex
November 16, 2019 at 5:58 pm · Reply

You could define it inline with the class definition (because it's trivial).



hellmet
November 17, 2019 at 12:01 am · Reply

Okay!

Parsa
October 9, 2019 at 3:13 pm · Reply

what about returning just by reference? Not const reference.
Is the object returned an r-value? So you can't return it by reference?

nascardriver
October 10, 2019 at 6:01 am · Reply

Most of the time you don't want to let the caller override the returned object. Returning a non-const reference allows the caller to do just that.

« Older Comments 1 2