13.6 — Overloading unary operators +, -, and !

ALEX · JANUARY 24, 2022

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:

```
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 -m_cents; // since return type is a Cents, this does an implicit conversion from int to Cents using the Cents(int) constructor
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 }
```

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:

```
1  #include <iostream>
2
3  class Point
4  {
5  private:
6      double m_x {};
7      double m_y {};
8      double m_z {};
9
10 public:
11     Point(double x=0.0, double y=0.0, double z=0.0):
12         m_x{x}, m_y{y}, m_z{z}
13     {
14     }
15
16     // Convert a Point into its negative equivalent
17     Point operator- () const;
18
19     // Return true if the point is set at the origin
20     bool operator! () const;
21
22     double getX() const { return m_x; }
23     double getY() const { return m_y; }
24     double getZ() const { return m_z; }
25 };
26
27 // Convert a Point into its negative equivalent
28 Point Point::operator- () const
29 {
30     return { -m_x, -m_y, -m_z };
31 }
32
33 // Return true if the point is set at the origin, false otherwise
34 bool Point::operator! () const
35 {
36     return (m_x == 0.0 && m_y == 0.0 && m_z == 0.0);
37 }
38
39 int main()
40 {
41     Point point{}; // use default constructor to set to (0.0, 0.0, 0.0)
42
43     if (!point)
44         std::cout << "point is set at the origin.\n";
45     else
46         std::cout << "point is not set at the origin.\n";
47
48     return 0;
49 }
```

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

Next lesson
13.7 Overloading the comparison operators

Back to table of contents

Previous lesson
13.5 Overloading operators using member functions

137 COMMENTS | Newest

furat · January 22, 2022 1:46 pm

Hello
the following program doesn't seem to work for me unless I put -f in between parenthesis `(-f).print();` Why?

```
#include <iostream>

class Fraction
{
private:
    int m_num{};
    int m_den{};
public:
    Fraction(int num, int den = 1) : m_num{ num }, m_den{ den }
    {}


    void print() const
    {
        std::cout << m_num << "/" << m_den << '\n';
    }
    Fraction operator-() const;
};

Fraction Fraction::operator-() const
{
    return Fraction{ -m_num, -m_den };
}
int main()
{
    const Fraction f{ 2, 5 };
    -f.print();

    return 0;
}
```

error: expression must have arithmetic or unscoped enum type.

with the first code in this tutorial, I was able to remove the parentheses (line 26 -nickle) and it worked just fine.

0 Reply

> **Alex** Author · Reply to furat · January 24, 2022 11:10 am
>
> `Operator.` has higher precedence than `operator-`, so this evaluates as `-(f.print())`. Since Fraction::print() returns a void, the compiler will complain about an illegal operand on void.
>
> You need the parenthesis to ensure that `-f` is evaluated first, and then `print()` invoked on the result.
>
> 1 Reply

>> **Furat** · Reply to Alex · January 25, 2022 12:10 am
>>
>> Thanks a lot. Appreciate it
>>
>> 0 Reply

Mateusz Kacpersi · January 22, 2022 1:19 am

Hello :)

```
Point Point::operator- () const
{
    return Point { -m_x, -m_y, -m_z };
}
```

but here return type doesn't have our class name why ?

```
Cents Cents::operator-() const
{
    return {-m_cents};
}
```

Last edited 1 month ago by Mateusz Kacpersi

0 Reply

> **Alex** Author · Reply to Mateusz Kacpersi · January 22, 2022 5:09 pm
>
> I've removed the explicit return type from the Point example. The compile can deduce the return type (from the function prototype), so we don't need to be explicit about it.
>
> In the Cents case, the {} are technically extraneous, as we don't need them to specify a single value. In the Point case, we need them to specify a list of initialization values.
>
> 0 Reply

> **omer** · Reply to Mateusz Kacpersi · January 22, 2022 1:10 pm
>
> I think m_cents is just our int type member variable. that's what I understood from the comment.
> I don't know what is the point of the braces though. why not just type `return m_cents` instead of `return {m_cents}`. and how is the constructor is involved in all of that? that is what confuses me.
>
> *Last edited 1 month ago by omer*
>
> 0 Reply

>> **Alex** Author · Reply to omer · January 24, 2022 10:54 am
>>
>> The function returns a Cents object, but we are returning an int value (m_cents), so the compiler will look for a constructor that it can use to construct a Cents from an int. The return value is used as the initializer.
>>
>> The braces are optional when providing a single value. I'll remove them.
>>
>> 0 Reply

Mateusz Kacpersi · January 20, 2022 12:09 pm

:)

Last edited 1 month ago by Mateusz Kacpersi

0 Reply

Mateusz Kacpersi · January 20, 2022 12:05 pm

Hello lads :)

I have one small question...

what's different between:

```
Cents( int cents )
 : m_cents( cents )
{
  //code...
}
```

and:

```
Cents( int cents )
 : m_cents{ cents }
{
  //code...
}
```

Like always massive THANK YYOU for your job here <3 God bless you.

Last edited 1 month ago by Mateusz Kacpersi

0 Reply

> **Alex** Author · Reply to Mateusz Kacpersi · January 20, 2022 10:10 pm
>
> The former uses direct (parenthesis) initialization for m_cents, the latter uses brace initialization. The latter is preferred.
>
> 0 Reply

RKay · November 15, 2021 9:49 am

```
Line 30 return Point(-m_x, -m_y, -m_z);
```

does not return a -point. I could not figure out but instead it gives (-m_z, 0, 0)
However, the following works

```
return Point{-m_x, -m_y, -m_z};
```

0 Reply

> **Alex** Author · Reply to RKay · November 16, 2021 10:32 am
>
> I retested the code, and operator- successfully returned (-m_x, -m_y, -m_z) as intended. That said, I updated it to use braces anyway.
>
> 0 Reply

Taylor7500 · October 24, 2021 11:11 am

Quick question, when using floating point types and overloading operator-(), what is the best way to protect against "-0" copping up when the input is zero? I'm doing something rather similar to your point example but as far as I can see the same thing will happen there when you ask for -Point if any of the x,y,z, are 0.

I know why it's happening and have seen a few different ideas (epsilon functions and adding 0.0 to everything) but neither seem like a simple and easily readable way to defend against negative zero in the single case where there is a 0 input and not when there isn't.

0 Reply

Tejero Joshua · August 16, 2021 4:44 pm

does (!point) favor our overloaded operator! on this statement
"
if (!point)
std::cout << "point is set at the origin.\n";
else
std::cout << "point is not set at the origin.\n";
"

0 Reply

> **Alex** Author · Reply to Tejero Joshua · August 17, 2021 11:10 am
>
> Yes.
>
> 0 Reply

Inevitable · August 8, 2021 12:33 am

Why we define operator- in such a way that it applies to the object's copy and leaves the current object untreated? why we don't apply it on current object ?

1 Reply

> **Alex** Author · Reply to Inevitable · August 8, 2021 4:26 pm
>
> Because if you had an expression like this:
>
> ```
> int x { -y };
> ```
>
> You wouldn't expect y to be changed.
>
> 5 Reply

Waldo Lemmer · July 21, 2021 1:14 am

- First code block, line 9 should use a member initializer list
- Last code block (before quiz), line 34:

```
return (m_x == 0.0 && m_y == 0.0 && m_z == 0.0)
```

Is comparison with 0.0 considered safe in this case?

0 Reply

> **Alex** Author · Reply to Waldo Lemmer · July 22, 2021 2:53 pm
>
> Yes, 0.0 can be represented exactly, so direct comparison with 0.0 is safe so long as you've directly initialized/assigned the 0.0 values. It might not be safe if you've done a bunch of math on the variables first and incurred some rounding error.
>
> 1 Reply

>> **Waldo Lemmer** · Reply to Alex · July 22, 2021 10:27 pm
>>
>> Thanks :)
>>
>> 0 Reply

Imagine · June 13, 2021 9:51 pm

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

    return 0;
}
```

Why was () used to initialize nickle instead of {} here?

0 Reply

> **Alex** Author · Reply to Imagine · June 14, 2021 11:53 am
>
> The lesson was written before brace-initialization was a thing. I've updated it to use braces.
>
> 0 Reply

>> **Rakesh** · Reply to Alex · June 30, 2021 4:06 am
>>
>> Hi,
>>
>> But as far as i understand, its better to use a list initialization as that would catch any narrowing conversions otherwise?
>>
>> 0 Reply

>>> **Alex** Author · Reply to Rakesh · July 4, 2021 2:17 pm
>>>
>>> Yes. Brace-initialization and list initialization are the same thing.
>>>
>>> 0 Reply

> **Imagine** · Reply to Imagine · June 13, 2021 9:53 pm
>
> Also, why is this
>
> ```
> Cents Cents::operator-() const
> {
>     return Cents(-m_cents);
> }
> ```
>
> and this
>
> ```
> Cents Cents::operator-() const
> {
>     return -m_cents;
> }
> ```
>
> same, which is better?
>
> 0 Reply

>> **Alex** Author · Reply to Imagine · June 14, 2021 11:54 am
>>
>> They are the same. In the top one, we're being explicit about the fact that we're returning a Cents object. In the bottom one, it's implicit.
>>
>> I've updated to use the bottom one since the extra verbosity doesn't really add anything.
>>
>> 0 Reply

>>> **JamesC** · Reply to Alex · July 20, 2021 1:58 am
>>>
>>> It looks quite confusing, like you're returning an int, though. In my opinion, you should at least point this out. Or write it as return { -m_cents }; to at least make it look like you're returning an anonymous object as opposed to just an int.
>>>
>>> 0 Reply

>>> **Imagine** · Reply to Alex · June 16, 2021 3:59 am
>>>
>>> Thanks, that was a small thing but it was really eating me.
>>>
>>> 0 Reply