9.4 — Overloading operators using member functions

BY ALEX ON OCTOBER 11TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In lesson **9.2 -- Overloading the arithmetic operators using friend functions**, you learned how to overload the arithmetic operators using friend functions. You also learned you can overload operators as normal functions. Many operators can be overloaded in a different way: as a member function.

Overloading operators using a member function is very similar to overloading operators using a friend function. When overloading an operator using a member function:

- The overloaded operator must be added as a member function of the left operand.
- The left operand becomes the implicit *this object
- All other operands become function parameters.

As a reminder, here's how we overloaded operator+ using a friend function:

```
#include <iostream>

class Cents
{
private:
    int m_cents;

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

    // Overload Cents + int
    friend Cents operator+(const Cents &cents, int value);

    int getCents() { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &cents, int value)
{
    return Cents(cents.m_cents + value);
}

int main()
{
    Cents cents1(6);
    Cents cents2 = cents1 + 2;
    std::cout << "I have " << cents2.getCents() << " cents.\n";

    return 0;
}
```

Converting a friend overloaded operator to a member overloaded operator is easy:

1. The overloaded operator is defined as a member instead of a friend (Cents::operator+ instead of friend operator+)
2. The left parameter is removed, because that parameter now becomes the implicit *this object.
3. Inside the function body, all references to the left parameter can be removed (e.g. cents.m_cents becomes m_cents, which implicitly references the *this object).

Now, the same operator overloaded using the member function method:

```
#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 + int
12      Cents operator+(int value);
13
14      int getCents() { return m_cents; }
15  };
16
17  // note: this function is a member function!
18  // the cents parameter in the friend version is now the implicit *this parameter
19  Cents Cents::operator+(int value)
20  {
21      return Cents(m_cents + value);
22  }
23
24  int main()
25  {
26      Cents cents1(6);
27      Cents cents2 = cents1 + 2;
28      std::cout << "I have " << cents2.getCents() << " cents.\n";
29
30      return 0;
31  }
```

Note that the usage of the operator does not change (in both cases, `cents1 + 2`), we've simply defined the function differently. Our two-parameter friend function becomes a one-parameter member function, with the leftmost parameter in the friend version (cents) becoming the implicit *this parameter in the member function version.

Let's take a closer look at how the expression `cents1 + 2` evaluates.

In the friend function version, the expression `cents1 + 2` becomes function call operator+(cents1, 2). Note that there are two function parameters. This is straightforward.

In the member function version, the expression `cents1 + 2` becomes function call cents1.operator+(2). Note that there is now only one explicit function parameter, and cents1 has become an object prefix. However, in lesson **8.8 -- The hidden "this" pointer**, we mentioned that the compiler implicitly converts an object prefix into a hidden leftmost parameter named *this. So in actuality, cents1.operator+(2) becomes operator+(&cents1, 2), which is almost identical to the friend version.

Both cases produce the same result, just in slightly different ways.

So if we can overload an operator as a friend or a member, which should we use? In order to answer that question, there's a few more things you'll need to know.

Not everything can be overloaded as a friend function

The assignment (=), subscript ([]), function call (()), and member selection (->) operators must be overloaded as member functions, because the language requires them to be.

Not everything can be overloaded as a member function

In lesson **9.3 -- Overloading the I/O operators**, we overloaded operator<< for our Point class using the friend function method. Here's a reminder of how we did that:

```
#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)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directl
y.
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}

int main()
{
    Point point1(2.0, 3.0, 4.0);

    std::cout << point1;

    return 0;
}
```

However, we are not able to overload operator<< as a member function. Why not? Because the overloaded operator must be added as a member of the left operand. In this case, the left operand is an object of type std::ostream. std::ostream is fixed as part of the standard library. We can't modify the class declaration to add the overload as a member function of std::ostream.

This necessitates that operator<< be overloaded as a normal function (preferred) or a friend.

Similarly, although we can overload operator+(Cents, int) as a member function (as we did above), we can't overload operator+(int, Cents) as a member function, because int isn't a class we can add members to.

Typically, we won't be able to use a member overload if the left operand is either not a class (e.g. int), or it is a class that we can't modify (e.g. std::ostream).

When to use a normal, friend, or member function overload

In most cases, the language leaves it up to you to determine whether you want to use the normal/friend or member function version of the overload. However, one of the two is usually a better choice than the other.

When dealing with binary operators that don't modify the left operand (e.g. operator+), the normal or friend function version is typically preferred, because it works for all parameter types (even when the left operand isn't a class object, or is a class that is not modifiable). The normal or friend function version has the added benefit of "symmetry", as all operands become explicit parameters (instead of the left operand becoming *this and the right operand becoming an explicit parameter).

When dealing with binary operators that do modify the left operand (e.g. operator+=), the member function version is typically preferred. In these cases, the leftmost operand will always be a class type, and having the object being modified become the one pointed to by *this is natural. Because the rightmost operand becomes an explicit parameter, there's no confusion over who is getting modified and who is getting evaluated.

Unary operators are usually overloaded as member functions as well, since the member version has no parameters.

The following rules of thumb can help you determine which form is best for a given situation:

- If you're overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function.
- If you're overloading a unary operator, do so as a member function.
- If you're overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function (preferred) or friend function.
- If you're overloading a binary operator that modifies its left operand, but you can't modify the definition of the left operand (e.g. operator<<, which has a left operand of type ostream), do so as a normal function (preferred) or friend function.
- If you're overloading a binary operator that modifies its left operand (e.g. operator+=), and you can modify the definition of the left operand, do so as a member function.

 **9.5 -- Overloading unary operators +, -, and !**

 **Index**

 **9.3 -- Overloading the I/O operators**

C++ TUTORIAL | PRINT THIS POST

57 comments to 9.4 — Overloading operators using member functions

David
January 17, 2020 at 7:39 pm · Reply

Hello,I still don't understand the meaning "The overloaded operator must be added as a member function of the left operand."
what is that meaning "of the left operand"?
Can you explain it in a simple way ?
Thanks for replying

nascardriver
January 18, 2020 at 1:25 am · Reply

```
1 a + b
```

where `a` is of type `A` and `b` is of type `B`.
`a` is the left operand, `b` is the right operand. `operator+` must be overloaded in class `A` for the above statement to work.

David
January 18, 2020 at 6:33 am · Reply

So you mean the return type for the operator+() should be the same as the type of the left operand.
Does it correct?

nascardriver
January 18, 2020 at 7:21 am · Reply

The `operator+` function must be a member of `A`. The return type doesn't matter.

```
class A
{
  // ...
  Something operator+(const B&b) { /* ... */ }
  // ...
};
```

Ged
January 6, 2020 at 2:30 pm · Reply

Question number 1

Why does this code compile. Shouldn't there be a compile error for too many operator+ functions with the same parameter?

```
#include <iostream>

class Cents
{
private:
    int m_cents;

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

    // Overload Cents + int
    Cents operator+(int value)
    {
        return Cents(m_cents + value);
    }
    friend Cents operator+(const Cents& n, int value);

    int getCents() { return m_cents; }
};

Cents operator+(const Cents& n, int value)
{
    return Cents(n.m_cents + value);
}
int main()
{
    Cents cents1(6);
    Cents cents2 = cents1 + 2;
    std::cout << "I have " << cents2.getCents() << " cents.\n";

    return 0;
```

```
32 }
```

nascardriver
January 8, 2020 at 5:50 am · Reply

Both functions can be defined, because they're different (member vs non-member), and both can be called

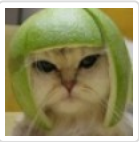
```
1 cents1.operator+(2); // member
2 operator+(cents1, 2); // free
```

When you do `cents1 + 2`, both functions are considered, but the member function wins. I can't tell you why the member function wins, maybe someone else can help.

Sekhar
May 16, 2019 at 7:13 am · Reply

Hi Alex,
Is that OK to state that "Objects that belong to libraries should be used only with the friend/normal functions like iostream objects where the left operand is modified and returned". If the statement holds good I think it can be added as a rule of thumb.

Thanks in advance,
Sekhar

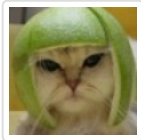


Alex
May 17, 2019 at 9:00 pm · Reply

Yes, there's no other alternative. You can't overload them as members.

sekhar
May 18, 2019 at 6:13 pm · Reply

Thanks Alex.Just want to mention that it's better to explicitly add the statement "If you’re overloading a binary operator that modifies its left operand (e.g. operator<<, operator >>) that belongs to standard library do so as a friend function." to rule of thumb that you have mentioned as part of last paragraph.



Alex
May 20, 2019 at 10:13 am · Reply

Updated. Thanks for the suggestion!

Arno Adam
February 6, 2019 at 7:06 am · Reply

Just a question if I understood the mechanics correctly: These past few chapters I was wondering whether operator overloading would cause memory leaks, because a new object is created every time an overloaded operator is used, correct? So I thought these new objects would continue to exist in memory alongside the original objects. But then I realized that the newly created object usually only has expression scope, because it's an r-value, right? Unless we assign it to something, like this:

```
1 someObjectVariable = object1 + object2; // using an overloaded + operator
```

nascardriver
February 6, 2019 at 8:48 am · Reply

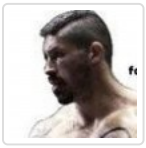
Hi Arno!

> a new object is created every time an overloaded operator is used
Only some operator create copies (eg. the binary + and - operators). Others don't (eg. insertion, extraction, +=, -=).
Those which create copies don't behave different than regular functions (Which also don't leak).
Memory leaks only occur when you're dynamically allocating memory and not freeing it.

Aron
April 3, 2018 at 8:51 pm · Reply

"Because the overloaded operator must be added as a member of the left operand"

We didn't need to modify the class std::ostream if we overload << operator by friend function.
However, if we want to overload << operator by member function, why do we have to modify the class std::ostream□which make it impossible□?



Adrian
January 5, 2018 at 6:50 am · Reply

Hello,
I need some help please...you sead

"Similarly, although we can overload operator+(Cents, int) as a member function (as we did above), we can’t overload operator+(int, Cents) as a member function, because int isn’t a class we can add members to."

But if i create an operator+(int, Cents) it works..

```
1  class Cents
2  {
3  private:
4      int m_cents;
5  public:
6      Cents(int cents) { m_cents = cents; }
7      Cents operator+(int value);
8      int getCents() { return m_cents; }
9
10     int operator+(Cents& c);//just works
11 };
12
13 Cents Cents::operator+(int value)
14 {
15     return Cents(m_cents + value);
16 }
17
18 int Cents::operator+(Cents & c)//works
19 {
20     return 0;
21 }
22
23 int main()
24 {
25     Cents cents1(6);
26     Cents cents2 = cents1 + 2;
27     std::cout << "I have " << cents2.getCents() << " cents.\n";
```

```
28
29      return 0;
30  }
```

Alex
January 7, 2018 at 1:21 pm · Reply

You didn't create operator+(int, Cents), you created Cents::operator+(Cents), which adds two Cents and returns an int.

Your code in main calls the (Cents + int) version, not the (int + Cents) version you thought you'd implemented.

KOBE JAPONICA
August 2, 2017 at 1:04 am · Reply

Rules of thumb to determine which form is best are to the point! Very good advice.
By the way, regarding the following sentence "However, we are not able to overloaded operator<< as a member function.", "overloaded" could be replaced by "overload"?

Omri
June 2, 2017 at 11:06 pm · Reply

Right, thanks.
Accordingly, perhaps , depending on compiler, we are not actually overloading an exiting function when we create our user defined operatorOPR(...) function , although it may be helpful to think of the process in these terms.
Thank you again.

Omri
June 2, 2017 at 5:36 am · Reply

Thank you for the clarification.
Is there a "straight forward" reason why build-in operators must be called using the "standard way" only?
Knowing a reason may some times assist in memorising a rule...

Alex
June 2, 2017 at 11:48 am · Reply

My best guess is because the compiler is free to not treat a built-in operator as a function at all -- it can just inline the resulting code, so there's no function with that name to even call.

omri
June 1, 2017 at 9:57 pm · Reply

Thank you for the reply.
Is it correct to say the following:
Overloaded versions of operatorOPR(...) functions can be created by the user and subsequently called using "x OPR y" format or more generally OPR format as suitable.
Executing OPR by calling operatorOPR(...) function (built in version or overloaded version) is never allowed.

Did I get this correct?
If yes, its a wonder why this restriction is necessary...

Alex
June 1, 2017 at 11:28 pm · Reply

For user-defined overloads of operators, you can call them via the operator name (e.g. operator+(x, y)) or via the standard way (x + y). For build-in operators, you must use the standard way.

Omri
May 31, 2017 at 8:55 pm · Reply

Hello Alex,
I understand that n+m implicitly stands for operator+(n,m). If n and m are integers (short, double etc.) an appropriate operator+(...) function is readily available and the operation is executed with a correct result returned. If n and/or m are user defined types an appropriate over loading of the operator+(...) function needs to be introduced by the user so that the
"+" operation will function as the user wishes when used with his "user introduced type"...
I tried to test the assumption that a function operator+(int x, int y) is indeed implicitly available and instead of writing:
int x=5, y=10, z; z=x+y;
I wrote:
int x=5, y=10, z; z=operator+(x,y);
The later did not compile.
Can you clarify?

Alex
June 1, 2017 at 9:19 pm · Reply

operator+(int, int) is a built-in operator. Built-in operators can't be called directly as functions.

dazedbrain
January 23, 2017 at 2:34 am · Reply

typo:
So if we can overload an operator as a friend or a member, which should use use?

thx for the awesome tutorials!

nelson
December 15, 2016 at 11:21 am · Reply

using friend function first argument is taken as const. but using memeber function why not const is taken?

Note: i am using chain of operators

using friend function operator overloading required const why?
obj1+obj2+obj3;

using member function operator overloading works even const is neglected why?
obj1+obj2+obj3;

Alex
December 15, 2016 at 3:22 pm · Reply

You can't pass a temporary (anonymous) result to a function by non-const reference, so in the member function version where the temporary result is being passed in as a parameter to the function, it needs to be const.

But you can call a non-const member function on a temporary result. So in the member function case, the const isn't needed. Though, this member function version of operator+ should really be const anyway, since it doesn't actually modify the class.

vaibhav
December 13, 2016 at 2:59 am · Reply

alex can u debug this program plz

```
/*wap in oop which overload binary '+' operator using member function*/
#include<iostream.h>
#include<conio.h>
class binary
{
float x;
float y;
public:
void collect()
{
cout<<"\n\tenter value for addition";
cin>>x>>y;
}
binary operator+(binarya,binaryb);
void display();
};
binary binary::operator+(binary B)
{
binary temp;
temp.x=x+B.x;
temp.y=y+B.y;
return(temp);
}
void binary::display()
{
cout<<x<<"+"<<y<<"\n";
}
void main()
{
binary B1,B2,B3;
cout<<"\n\t----------input----------";
B1.collect();
B2.collect();
B3=B1.operator+(B2);
cout<<"\n\tB1.display";
cout<<"\n\tB2.display";
cout<<"\n\t-------------------------------------";
cout<<"\n\tB3";B3.display();
```

```
cout<<"\n\t-----------------------------------";
getch();
}
```

Alex
December 14, 2016 at 10:44 am · Reply

This line of your class definition doesn't match your member function implementation:

```
binary operator+(binarya,binaryb);
```

You probably meant:

```
binary operator+(binary B);
```

Vasant Prabhu
November 5, 2016 at 9:57 pm · Reply

Hi, I expected my code to return 12. But its giving me 5 as result.
Not sure why this is happening.

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

class Cents
{
    private:
        int m_cents;
    public:
        Cents(int myval=0){
            m_cents = myval;
        }
        int getCents()const{return m_cents;}
        Cents operator+=(int value);
};

Cents Cents::operator+=(int value)
{
    return Cents(m_cents+value);
}
int main()
{
    Cents c1(5);
    c1+=7;
    cout << "result is"<<c1.getCents();
    return 0;
}
```

Alex
November 6, 2016 at 8:45 pm · Reply

Your += operator is incorrect. += should modify the implicit object and then return that. Instead, you're creating a new Cents object and returning that.

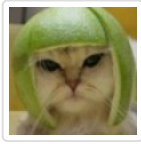
So when you do c1+=7, operator c1.operator+=(7) is returning a new Cents with the value 12 that is being discarded.

Try this instead:

```
Cents& Cents::operator+=(int value) // note that we're returning a reference
{
    m_cents += value;
    return *this;
}
```

methen
November 11, 2016 at 9:39 am · Reply

can't return type be void??



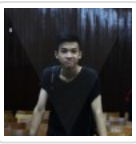
Alex
November 13, 2016 at 1:13 pm · Reply

It can return void, but then you won't be able to chain operators, like this:

```
a = b += 5; // add 5 to b and assign that value to a
```

methen
November 20, 2016 at 7:42 am · Reply

oh thanks.

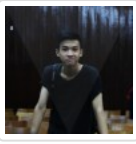


mrK
May 6, 2016 at 8:58 am · Reply

```
#include <string>
#include <iostream>

class Cents
{
private:
    int m_nCents;
public:
    Cents(int cents) {m_nCents = cents;}

    friend Cents operator+(const Cents &cents, int value);
    int getCents()
    {
            return m_nCents;
    }
}

Cents operator+(const Cents &cents, int value)
{
    return Cents(cents.m_nCents + value)
}

int main()
{
    Cents cents1(6);
    Cents cents2 = cents1 + 2;
    std::cout << "I have " << cents2.getCents() << " cents \n";

```

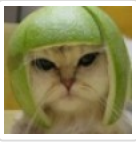
```
29      return 0;
30  }
```

hey, when i used ubuntu 14.04 to run that code, some errors make me confused "error: expected initializer before 'operator'". ???



mrK
May 6, 2016 at 9:20 am · Reply

and error in line 18



Alex
May 8, 2016 at 7:07 pm · Reply

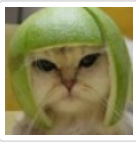
This is one of the many seemingly random errors that can occur when you forget the semicolon at the end of a class declaration.

Josh W
April 30, 2016 at 10:30 pm · Reply

I'm really starting to get the Point!

I've been following these tutorials for a few days now and am learning a lot more than I've learned anywhere else. I took a programming class in HS (it taught VB.NET) and I realize now that the teacher didn't know anything about programming (we never talked about Object Oriented Programming for starters). Even so, I've been teaching myself to program in VB since then (about 8 years now) and learned quite a bit but came to a point where I decided I needed to try out C++ and find out why it's all the rage with the kids. Every tutorial I turned to started out expecting me to already be an expert or quickly went there. This is the only tutorial I've found that has given me genuine instruction in C++ that has given me the confidence to tackle my own projects in this new (to me) language!

```
1  std::cout << "Thanks Alex!" << 'n';
```



Alex
May 1, 2016 at 6:52 pm · Reply

You're welcome. Thanks for visiting.

The Vip
April 25, 2016 at 5:52 am · Reply

Bro, there you have given link for section 9.6 , you missed 9.5 :-)

BlueTooth4269
February 21, 2016 at 9:21 am · Reply

How would we overload the + operator for adding two Cents objects together as a member function?
Is it even possible/recommended?

Alex
February 21, 2016 at 3:54 pm · Reply

It's possible -- you'd do it the same way as in the Cents + int case above, but with the function taking a Cents parameter instead of an int.

However, it's recommended to use the friend version, as it's both more intuitive, and more flexible (you can implement int + Cents as a friend, but not as a member, since the left operand must be a class type in the member version, and int isn't).

Rahul
November 15, 2015 at 3:44 am · Reply

"Remember that when C++ sees the function prototype Cents Cents::operator-();, the compiler internally converts this to Cents operator-(const Cents *this), which you will note is almost identical to our friend version Cents operator-(const Cents &cCents)!"

Here, function prototype is Cents operator-();
And if you mean to say its Cents Cents::operator-(); - which is defined outside outside the class, then the compiler would internally convert it to Cents Cents::operator-(const Cents *this) as it is a member function.

Please clarify how compiler internally converts and links prototype and definition.

Arun Mathew Iype
October 25, 2015 at 5:54 am · Reply

What is the reason that the operator= should always be a member function? Can this not be done using a friend function too?

Alex
October 25, 2015 at 1:06 pm · Reply

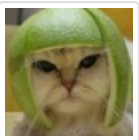
C++ requires operator= to be a member function. I'm guessing that this is because C++ will auto-generate one for you if you don't define one. If you were able to define one outside the class, the compiler wouldn't know whether to call the auto-generated one, or the one you wrote outside the class.

Vishnu
October 4, 2015 at 6:21 am · Reply

Why Operator is not able to modify the private values when overloaded as function 1(defined below) ?

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

class abc{
    int x,y,z;
    public:
        //abc():x(1),y(2),z(3){};
        abc(int a = 1 , int b = 2, int c = 3){
            x=a;
            y=b;
            z=c;
        }

        abc set_value(int a,int b, int c){
            x=a;
            y=b;
```

```
            z=c;
        }

        abc operator+(int data);
        abc operator-();
        friend ostream& operator<< (ostream &,const abc &);
        friend istream& operator>>(istream &,abc &);
};

ostream& operator<<(ostream &out ,const abc &o1){
    cout << "x :" << o1.x << endl << "y :" << o1.y << endl << "z : " << o1.z << endl ;
    return out ;
}

istream& operator>>(istream &in , abc &o1){
    cin >> o1.x >> o1.y >> o1.z ;
    return in ;
}
/*****************************DOUBT AREA*************************************/
// Use only one of the below.

//Function 1 ;
abc abc::operator+(int data){
    cout << data << endl ;
    return abc(x+ data,y + data,z + data);
}

//Function 2;
abc abc::operator+(int data){
    cout << data << endl ;
    return set_value(data+x,y + data,z + data);
}

/*********************************DOUBT AREA*********************************/

int main(){

    abc o1;
    cin >> o1;
    cout << "The values are -" << o1 << endl;
    o1+(10);
    cout << o1;
}
```



Alex
October 5, 2015 at 8:30 am · Reply

Function 1 is correct. Consider your code:

```
o1+(10);
```

This takes object o1 and adds 10, which calls abc::operator+(int), which then returns the new object. However, your code does not do anything with this object, so it is discarded.

You probably intended to do this:

```
o1 = o1 + 10;
```

That way, the return value of o1 + 10 is assigned back to o1.

Function 2 is logically incorrect. operator+ should not modify the object being called (operator += should).

Vishnu
October 5, 2015 at 11:03 am · Reply

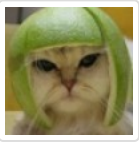
Thank you very much :)

Joseph
March 21, 2015 at 1:07 am · Reply

Regarding the following statement:

> "However, when dealing with operands that modify the class itself (eg. operators =, +=, -=, ++, --, etc…) the member function method is typically used because C++ programmers are used to writing member functions (such as access functions) to modify private member variables."

Would it be correct to say that C++ programmers mainly use friend functions to set and modify private member variables, and member functions to get and compare private member variables?



Alex
April 5, 2016 at 3:38 pm · Reply

No.

Member functions are generally preferred whenever access to private members are needed. Friend functions are typically used when a function needs access to private data but a member function can't be used (e.g. a member of one class needs access to the private data of another class). Like many things, there are exceptions. One exception is for binary operator overloads that don't modify the left operand.

Priyali
September 17, 2013 at 12:13 pm · Reply

Hi Alex,

I tried overloading binary addition (+) operator using a member function as shown below.

```
using namespace std;

class Cents
{
private:
int m_nCents;

public:
Cents(int nCents = 0) { m_nCents = nCents; }

// Overload for Cents + Cents
Cents operator+(const Cents &cTemp);

int GetCents() { return m_nCents; }
};

// note: this function is a member function!
Cents Cents::operator+(const Cents &cTemp)
```

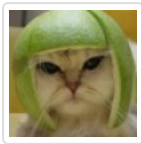
```
{
return Cents(m_nCents + cTemp.m_nCents);
}

int main()
{
Cents cAdd;
Cents c1(4);
Cents c2(6);
cAdd = c1 + c2;
cout << "I have " << cAdd.GetCents() << " cents." << endl;

return 0;
}
```

The code is compiling without errors and giving me the desired output. As per my understanding, when I do **c1+c2** , the call gets converted into **Cents operator+(const Cents *this, const Cents &cTemp)**. How could "this pointer" which points to object c1 can access private member m_nCents of object c2(cTemp) even if its not a friend function. This is a bit confusing for me.

egordo19
September 28, 2014 at 4:44 pm · Reply

it is a member function,
this is because of the line...

Cents Cents::operator+(const Cents &cTemp)



Alex
April 5, 2016 at 3:30 pm · Reply

Exactly. Your overloaded operator is a member function, and member functions can access private data.

madhukar
August 30, 2012 at 10:17 am · Reply

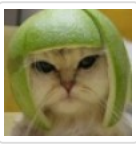
i want to know if i can have an overloaded unary operator. (don't know if that even makes sense.)
i don't see how it is possible. The problem i was thinking about was-
suppose obj1 and obj2 are two objects of the same class.
i want to do:
-M1; // the operator - function returns no value, just changes the values of the data members of M1
M2=-M1 // the operator - function returns an object of the class.
i was thinking of using both member function and friend function together would allow me to do this, haven't tried it yet though.
Thanks,
madhukar

madhukar
August 30, 2012 at 10:19 am · Reply

also, when operator - is called in
M2=-M1;

data members of M1 remain unchanged.

PS: im not trolling, was really wondering if its possible :S



Alex
April 5, 2016 at 3:28 pm · Reply

Yes, you can have an overloaded unary operator.

If you do M2 = -M1, the unary operator- will be applied to M1 (with the result being returned by value, so that M1 isn't modified), and then the result will be assigned to M2.

We will look at this in more detail shortly.

Tom
April 6, 2008 at 9:21 am · Reply

Thanks Alex, this is starting to make cents. ;)

Darren
June 21, 2016 at 2:41 am · Reply

Every time I read this joke I groan (usually internally) and do any eye roll. I though it worthy of comment.