# 9.2a — Overloading operators using normal functions

BY ALEX ON MAY 23RD, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In the previous lesson, we overloaded operator+ as a friend function:

```
#include <iostream>

class Cents
{
private:
    int m_cents;

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

    // add Cents + Cents using a friend function
    friend Cents operator+(const Cents &c1, const Cents &c2);

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

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents(c1.m_cents + c2.m_cents);
}

int main()
{
    Cents cents1(6);
    Cents cents2(8);
    Cents centsSum = cents1 + cents2;
    std::cout << "I have " << centsSum.getCents() << " cents." << std::endl;

    return 0;
}
```

Using a friend function to overload an operator is convenient because it gives you direct access to the internal members of the classes you're operating on. In the initial Cents example above, our friend function version of operator+ accessed member variable m_cents directly.

However, if you don't need that access, you can write your overloaded operators as normal functions. Note that the Cents class above contains an access function (getCents()) that allows us to get at m_cents without having to have direct access to private members. Because of this, we can write our overloaded operator+ as a non-friend:

```
#include <iostream>

class Cents
{
private:
    int m_cents;

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

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

```
};

// note: this function is not a member function nor a friend function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
        // we don't need direct access to private members here
    return Cents(c1.getCents() + c2.getCents());
}

int main()
{
    Cents cents1(6);
    Cents cents2(8);
    Cents centsSum = cents1 + cents2;
    std::cout << "I have " << centsSum.getCents() << " cents." << std::endl;

    return 0;
}
```

Because the normal and friend functions work almost identically (they just have different levels of access to private members), we generally won't differentiate them. The one difference is that the friend function declaration inside the class serves as a prototype as well. With the normal function version, you'll have to provide your own function prototype.

Cents.h:

```
class Cents
{
private:
    int m_cents;

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

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

// Need to explicitly provide prototype for operator+ so uses of operator+ in other files know
Cents operator+(const Cents &c1, const Cents &c2);
```

Cents.cpp:

```
#include "Cents.h"

// note: this function is not a member function nor a friend function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
        // we don't need direct access to private members here
    return Cents(c1.getCents() + c2.getCents());
}
```

main.cpp:

```
#include <iostream>
#include "Cents.h"

int main()
{
    Cents cents1(6);
    Cents cents2(8);
```

```
8       Cents centsSum = cents1 + cents2; // without the prototype in Cents.h, this would fail to
9       std::cout << "I have " << centsSum.getCents() << " cents." << std::endl;
10
11      return 0;
12  }
```

In general, a normal function should be preferred over a friend function if it's possible to do so with the existing member functions available (the less functions touching your classes's internals, the better). However, don't add additional access functions just to overload an operator as a normal function instead of a friend function!

*Rule: Prefer overloading operators as normal functions instead of friends if it's possible to do so without adding additional functions.*

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

 **Index**

 **9.2 -- Overloading the arithmetic operators using friend functions**

C++ TUTORIAL | PRINT THIS POST

## 30 comments to 9.2a — Overloading operators using normal functions

HoiNguyen
January 7, 2020 at 10:16 am · Reply

Hi Alex and nascardriver
I'm trying to overload operator in Derived class
But i got an error: no suitable constructor exists to convert from "double" to "Derived"
Can you help me pointing out what's wrong.

Thanks you!
My code below:

```
class Base
{
private:
    double m_Base;

public:
    Base(double pBase) : m_Base{pBase} {}
    Base() {}
    ~Base() {}
    double Get() const
    {
        return m_Base;
    }
};

Base operator+(const Base &pBase1, const Base &pBase2)
{
    return pBase1.Get() + pBase2.Get();
}

class Derived : public Base
{
private:
    double m_Derived;

public:
    double GetDerived() const
    {
        return m_Derived;
    }
    Derived(double pBase, double pDerived) : Base{pBase}, m_Derived{pDerived} {}
    Derived() {}
    ~Derived() {}
};

Derived operator+(const Derived &pDerived1, const Derived &pDerived2)
{
    return pDerived1.GetDerived() - pDerived2.GetDerived(); //This line get error -- no sui
}
```

nascardriver
January 8, 2020 at 3:53 am · Reply

`operator+` is declared to return a `Derived` (Line 36).
You're returning a `double` (Line 38) (A `double` minus a `double` is a `double`.).
A `double` is not a `Derived`, so a conversion must be made.
There is no conversion (ie. `Derived(double)`) to a `double`.
You get an error.
Add a constructor to `Derived` that takes a single `double` parameter.

HoiNguyen
January 8, 2020 at 4:27 am · Reply

Thanks nascardriver
I tried and it works

But i have another question.
As you say:
" `operator+` is declared to return a `Derived` (Line 36).
You're returning a `double` (Line 38) (A `double` minus a `double` is a `double`.).
A `double` is not a `Derived`, so a conversion must be made.
There is no conversion (ie. `Derived(double)`) to a `double`.
You get an error.
Add a constructor to `Derived` that takes a single `double` parameter. "

So why overloading operator at Base class doesn't get that error.
And i try edit my code as:

```
Derived operator+(const Derived &pDerived1, const Derived &pDerived2)
{
    double temp1 = pDerived1.Get() + pDerived2.Get();
    double temp2 = pDerived1.GetDerived() + pDerived2.GetDerived();
    return Derived(temp1, temp2);
}
```

And it's working. Is it potential any risk? Do you have any advice.
Thanks you!

nascardriver
January 9, 2020 at 3:08 am · Reply

`Base` has a constructor that takes a single `double`. This constructor will be used for the conversion. `Derived` didn't have such a constructor (And the `Base` constructor can't be used to create a `Derived`).
There's no issue with your new code. If it does what you want, it's correct. You should use brace initialization for higher type safety.

```
double temp{ /*...*/ };
```

HoiNguyen
January 9, 2020 at 3:45 am · Reply

I really appreciate your help. Thank you, nascardriver

Hadi
August 2, 2019 at 8:50 pm · Reply

```
#include <iostream>

class Cents
{
private:
    int m_cents;

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

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

// note: this function is not a member function nor a friend function!
```

```
15 Cents operator+(const Cents &c1, const Cents &c2)
16 {
17     // use the Cents constructor and operator+(int, int)
18      int x=(c1.getCents() + c2.getCents());
19      Cents a(0);
20 
21     return a(x);
22 }
23 
24 int main()
25 {
26     Cents cents1(6);
27     Cents cents2(8);
28     Cents centsSum = cents1 + cents2;
29     std::cout << "I have " << centsSum.getCents() << " cents." << std::endl;
30 
31     return 0;
32 }
```

hello Alex
What wrong with the code above ?
I don't want to return anonymous object
I want to return a normal object

**nascardriver**
August 3, 2019 at 1:45 am · Reply

Hello Hadi!

- Initialize your variables with brace initializers.
- Initialize members in the member initializer list.
- You can only call on constructor during initialization. Line 21 tries to call `Cents::operator()`, but that doesn't exist. Initialize `a` with `x` in line 19, or return immediately.

Hadi
August 4, 2019 at 4:14 pm · Reply

Thank you!
I didn't know that I can only call on constructor during initialization.
you are very experienced.

**Ruturaj Vaidya**
September 21, 2018 at 9:05 pm · Reply

Hello sir,

Why we use

```
1 return Cents(c1.getCents() + c2.getCents());
```

?

I believe

```
1 return (c1.getCents() + c2.getCents());
```

works just fine (At least when I tried).

Ruturaj

Alex
September 24, 2018 at 7:40 am · Reply

Both work, because the compiler is smart enough in your case to be able to infer we want to create an object of the type the function is returning. That said, it's usually better to be explicit about your intent wherever reasonable, so I'd still favor my version.

**Ruturaj Vaidya**
September 24, 2018 at 7:58 am · Reply

Thanks sir @Alex for your reply

CrazyL
August 4, 2017 at 6:23 am · Reply

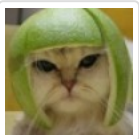
@Alex,

guess here's a typo:
- we'll generally won't differentiate them. --> we generally won't differentiate them

The following sentence I had to reread a few times before I understood:
- The one difference is that with the friend function, the friend function declaration inside the class also serves as a prototype.

Perhaps an easier sentence structure is in order, like
--> The one difference is that the friend function declaration inside the class serves as a prototype as well.

Apart from that, the the lesson was great!

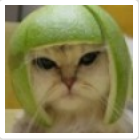


Alex
August 6, 2017 at 11:36 am · Reply

Updated as per your suggestions. Thanks!

omri
May 26, 2017 at 2:47 pm · Reply

According to my questions and replies from 8.5, I think the following rephrasing is due:
return Cents(c1.m_cents + c2.m_cents);
is not an explicit call to the Cents(int x) constructor, but rather a syntax to create an anonymous Cents object with the required m_cents member. This anonymous object is returned to the caller etc.



Alex
May 27, 2017 at 7:01 am · Reply

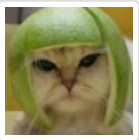
It's not a direct call to the Cents() constructor, but it does use the Cents() constructor, so saying we're "using" the constructor is appropriate.

Omri
May 24, 2017 at 9:26 pm · Reply

```
// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{// use the Cents constructor and operator+(int, int)
// we can access m_cents directly because this is a friend function
return Cents(c1.m_cents + c2.m_cents);
}
```

Is the following correct?

```
// note: this function definition is not a member function definition!
Cents operator+(const Cents &c1, const Cents &c2)
{/*lt uses the Cents constructor explicitly and can access objects c1, c2 m_cents member directly because this is a Cents friend function*/
return Cents(c1.m_cents + c2.m_cents);
}
```



Alex
May 25, 2017 at 1:44 pm · Reply

Yes.

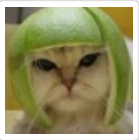
Omri
May 24, 2017 at 9:18 pm · Reply

In the first code section above:

```
// add Cents + Cents using a friend function
friend Cents operator+(const Cents &c1, const Cents &c2)
```

would it be correct to say here:

```
/*add Cents + Cents using a friend function named operator+
note that this function is not a member function of the class.
It is an "ordinary" function but special in the fact that it is given permission "by the class" to access this class's private members*/
friend Cents operator+(const Cents &c1, const Cents &c2)
```



Alex
May 25, 2017 at 1:44 pm · Reply

Yes.

John
January 20, 2017 at 11:31 am · Reply

Thank you for including an example using separate .h and .cpp files! Seeing the & in front of the parameters in the declaration

```
1 Cents operator+(const Cents &c1, const Cents &c2);
```

helped solve a problem for me! I was only including the & in my function definition in my .cpp file.

Connor
July 13, 2016 at 1:43 pm · Reply

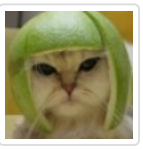
Hi Alex, why are we including the function prototype for

```
1 Cents operator+(const Cents &c1, const Cents &c2);
```

in the header file?
Since the .cpp file must be included in the solution is there anything wrong with defining the

```
1 Cents operator+(const Cents &c1, const Cents &c2);
```

only in the .cpp file?



Alex
July 14, 2016 at 4:18 pm · Reply

We put the forward declaration of operator+ in the header so that any file that #includes Cents.h can use that operator+ without additional work (other than compiling Cents.cpp into the project).

If you put the forward declaration in Cents.cpp, main.cpp won't see it there, since main.cpp can't see what's other code files while its being compiled.
If you put the forward declaration in main.cpp, it would work for this program. But in a larger program, you'd have to do the same for every file that wanted to use this function, and that's a pain.

Devaraj
July 13, 2016 at 4:10 am · Reply

Hi Alex,
In the first code above , overloading function is not using the access function getcents(), than how it can access private members ? Hope the code is wrong!

Alex
July 14, 2016 at 3:55 pm · Reply

If a class makes a function a friend, that function can access the private members of the class as if it were a member of the class itself. In this case, our overloaded operator+ is a friend of class Cents, so it can access private member m_cents directly.

That's the whole point of the lesson! :)

Devaraj
July 14, 2016 at 5:25 pm · Reply

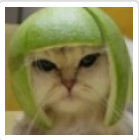
Oops! sorry, I didn't see the keyword freind there before, as the topic name was normal function.my bad

SUDARSHAN KOLAR
July 8, 2016 at 11:00 am · Reply

Hi Alex... really nice lectures... thanks a ton...
int getCents() const { return m_cents; }...

why are we using the _const_ keyword above? Can you also please link where was this part discussed previously in chapters?

Thanks
Sudarshan



Alex
July 8, 2016 at 11:51 am · Reply

This is explained in lesson **8.10 -- Const class objects and member functions**.

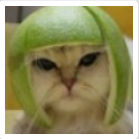
Darren
June 20, 2016 at 8:17 am · Reply

"...touching your classes' internals,..." there's a joke in there somewhere.

Shiva
May 28, 2016 at 3:35 am · Reply

This is the first time I'm seeing a chapter on this site without even a single comment, so I'll add one ;) :

> *Because of this, we can write our overloaded operator+ as a non-member:* (in this context *non-friend* would be more appropriate)



Alex
May 28, 2016 at 1:25 pm · Reply

That's because I just recently wrote this lesson. Thanks for the wording suggestion. Integrated.