9.1 — Introduction to operator overloading

BY ALEX ON SEPTEMBER 24TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In lesson **7.6 -- Function overloading**, you learned about function overloading, which provides a mechanism to create and resolve function calls to multiple functions with the same name, so long as each function has a unique function prototype. This allows you to create variations of a function to work with different data types, without having to think up a unique name for each variant.

In C++, operators are implemented as functions. By using function overloading on the operator functions, you can define your own versions of the operators that work with different data types (including classes that you've written). Using function overloading to overload operators is called **operator overloading**.

In this chapter, we'll examine topics related to operator overloading.

Operators as functions

Consider the following example:

```
int x = 2;
int y = 3;
std::cout << x + y << '\n';
```

The compiler comes with a built-in version of the plus operator (+) for integer operands -- this function adds integers x and y together and returns an integer result. When you see the expression `x + y`, you can translate this in your head to the function call `operator+(x, y)` (where operator+ is the name of the function).

Now consider this similar snippet:

```
double z = 2.0;
double w = 3.0;
std::cout << w + z << '\n';
```

The compiler also comes with a built-in version of the plus operator (+) for double operands. Expression w + z becomes function call `operator+(w, z)`, and function overloading is used to determine that the compiler should be calling the double version of this function instead of the integer version.

Now consider what happens if we try to add two object of a user-defined class:

```
Mystring string1 = "Hello, ";
Mystring string2 = "World!";
std::cout << string1 + string2 << '\n';
```

What would you expect to happen in this case? The intuitive expected result is that the string "Hello, World!" would be printed on the screen. However, because Mystring is a user-defined class, the compiler does not have a built-in version of the plus operator that it can use for Mystring operands. So in this case, it will give us an error. In order to make it work like we want, we'd need to write an overloaded function to tell the compiler how the + operator should work with two operands of type Mystring. We'll look at how to do this in the next lesson.

Resolving overloaded operators

When evaluating an expression containing an operator, the compiler uses the following rules:

- If *all* of the operands are fundamental data types, the compiler will call a built-in routine if one exists. If one does not exist, the compiler will produce a compiler error.
- If *any* of the operands are user data types (e.g. one of your classes, or an enum type), the compiler looks to see whether the type has a matching overloaded operator function that it can call. If it can't find one, it will try to convert one or more of the user-defined type operands into fundamental data types so it can use

a matching built-in operator (via an overloaded typecast, which we'll cover later in this chapter). If that fails, then it will produce a compile error.

What are the limitations on operator overloading?

First, almost any existing operator in C++ can be overloaded. The exceptions are: conditional (?:), sizeof, scope (::), member selector (.), member pointer selector (.*), typeid, and the casting operators.

Second, you can only overload the operators that exist. You can not create new operators or rename existing operators. For example, you could not create an operator ** to do exponents.

Third, at least one of the operands in an overloaded operator must be a user-defined type. This means you can not overload the plus operator to work with one integer and one double. However, you could overload the plus operator to work with an integer and a Mystring.

Fourth, it is not possible to change the number of operands an operator supports.

Finally, all operators keep their default precedence and associativity (regardless of what they're used for) and this can not be changed.

Some new programmers attempt to overload the bitwise XOR operator (^) to do exponentiation. However, in C++, operator^ has a lower precedence level than the basic arithmetic operators, which causes expressions to evaluate incorrectly.

In basic mathematics, exponentiation is resolved before basic arithmetic, so 4 + 3 ^ 2 resolves as 4 + (3 ^ 2) => 4 + 9 => 13.
However, in C++, the arithmetic operators have higher precedence than operator^, so 4 + 3 ^ 2 resolves as (4 + 3) ^ 2 => 7 ^ 2 => 49.

You'd need to explicitly parenthesize the exponent portion (e.g. 4 + (3 ^ 2)) every time you used it for this to work properly, which isn't intuitive, and is potentially error-prone.

Because of this precedence issue, it's generally a good idea to use operators only in an analogous way to their original intent.

Rule: When overloading operators, it's best to keep the function of the operators as close to the original intent of the operators as possible.

Furthermore, because operators don't have descriptive names, it's not always clear what they are intended to do. For example, operator+ might be a reasonable choice for a string class to do concatenation of strings. But what about operator-? What would you expect that to do? It's unclear.

Rule: If the meaning of an operator when applied to a custom class is not clear and intuitive, use a named function instead.

Within those confines, you will still find plenty of useful functionality to overload for your custom classes! You can overload the + operator to concatenate your user-defined string class, or add two Fraction class objects together. You can overload the << operator to make it easy to print your class to the screen (or a file). You can overload the equality operator (==) to compare two class objects. This makes operator overloading one of the most useful features in C++ -- simply because it allows you to work with your classes in a more intuitive way.

In the upcoming lessons, we'll take a deeper look at overloading different kinds of operators.

9.2 -- Overloading the arithmetic operators using friend functions


Index


8.x -- Chapter 8 comprehensive quiz

C++ TUTORIAL | PRINT THIS POST

36 comments to 9.1 — Introduction to operator overloading

mmp52
August 7, 2019 at 11:42 pm · Reply

Hi!
May you give me an example of a member pointer selector (.*) ?

Thanks!

nofinkski
April 30, 2019 at 1:30 am · Reply

Hi Alex,
first comment ever. So, thanks for the site. Great work!!

I wonder, how 4 + 3 ^ 2 evaluates to 49.

Imho, it evaluates like the follwing:
4 + 3 ^ 2
7 ^ 2
111b ^ 010b
101b
5

nascardriver
April 30, 2019 at 3:34 am · Reply

Alex uses ^ as the power operator in this lesson, not a bitwise XOR.

Chris
November 13, 2018 at 2:39 pm · Reply

"Some new programmers attempt to overload the bitwise XOR operator (^) to do exponentiation. However, in C++, operator^ has a lower precedence level than the basic arithmetic operators, which causes expressions to evaluate incorrectly.

In basic mathematics, exponentiation is resolved before basic arithmetic, so 4 + 3 ^ 2 resolves as 4 + (3 ^ 2) => 4 + 9 => 13.
However, in C++, the arithmetic operators have higher precedence than operator^, so 4 + 3 ^ 2 resolves as (4 + 3) ^ 2 => 7 ^ 2 => 49.

You'd need to explicitly parenthesize the exponent portion (e.g. 4 + (3 ^ 2)) every time you used it for this to work properly, which isn't intuitive, and is potentially error-prone."

Even if you parenthesized the exponent, wouldn't it still not work properly because, as you mentioned,"Third, at least one of the operands in an overloaded operator must be a user-defined type."?

Alex
November 13, 2018 at 11:04 pm · Reply

True. I use integers in the example because it's easy to show the precedence issue. But the presumption is that you'll be overloading operators for your own user-defined types, not trying to redefine operators for fundamental types.

Adrian
July 2, 2018 at 2:37 am · Reply

why instead of using built-in function for each type for + operator they just didn't define a template function to do that?

Alex
July 9, 2018 at 10:22 am · Reply

Templates still resolve to normal function calls once the parameter types are determined. Also if you had a template function that added an object of type T and type S together, would it return a T or an S?

Jujinko
August 10, 2017 at 11:09 pm · Reply

Hey Alex,
in the first two examples you wrote cout without the 'std::' as if you had called "using" before, which you didn't. The example is still understandable, but for consistency you might wanna change that.
Greetings, J

Alex
August 12, 2017 at 1:01 pm · Reply

Thanks! Fixed.

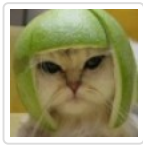
Mehul
April 14, 2017 at 11:44 pm · Reply

ALex this is the best site which i am following from 2014 to learn the concepts, Thanks a lot to make the concepts easily understandable :-)

Jim
April 4, 2017 at 7:27 pm · Reply

Hello Alex,

Could you please explain overloading of new and delete operator? I really appreciate any help that you can provide.



Alex
April 5, 2017 at 9:25 am · Reply

Overloading new and delete is an advanced topic and probably worthy of its own separate tutorial. I can't really do it justice in a response to a comment.

A
July 22, 2016 at 4:33 pm · Reply

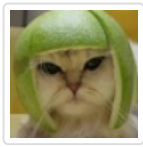
Do the next lessons build up on operator overloading ?
Little short on time :)

hero76
May 24, 2016 at 11:22 pm · Reply

nice work

NotAMeme
May 6, 2016 at 9:40 am · Reply

Whats the flight velocity of an unladen swallow?



Alex
May 8, 2016 at 7:08 pm · Reply

What do you mean, an African or European Swallow?

Darren
June 20, 2016 at 7:54 am · Reply

I don't know … Arrrgggghhhhh!!!

Matt
March 31, 2018 at 1:47 pm · Reply

This made my day.

mashariki
April 11, 2013 at 3:39 am · Reply

2013 and loving your site! C++ well explained and very easy to understand..have tried out other sites but this is the best..much thanks from nairobi.

peter
May 16, 2010 at 9:39 pm · Reply

Hi Alex,

Is there any reason why we cannot overload the following operators:

arithmetic if (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*)

Does any one have ideas?

p
October 9, 2010 at 1:23 am · Reply

no idea

jadon
June 28, 2015 at 10:00 pm · Reply

because overloading those operators wouldnt make a whole lot of sense. and some real ambiguity would occur between normally happens and the calling the overload method.

Alex
March 29, 2016 at 9:00 am · Reply

When Stroustrup was designing C++, he decided that operators that were close to the core of the language should not be overridable, because it would lead to ambiguity or confusion. His goal was to allow user-defined classes to work analogously to built-in types, not to allow you redefine how core language mechanisms work.

Fan
October 28, 2019 at 7:27 am · Reply

Is that list complete?

In https://www.tutorialspoint.com/operators-that-cannot-be-overloaded-in-cplusplus, I also found the typeid operator that cannot be overloaded.

Alex
October 28, 2019 at 10:08 am · Reply

Added typeid, as well as the casting operators. Thanks for pointing out the omission.

Renu
January 15, 2008 at 11:37 am · Reply

I didnt understand this
"all operators keep their current precedence and associativity, regardless of what they're used for. For example, the bitwise XOR operator (^) could be overloaded to do exponents, except it has the wrong precedence and associativity and there is no way to change this "

Could you please elaborate this?
Thanks,
Renu

Alex
January 15, 2008 at 12:30 pm · Reply

Consider the following mathematical equation: 5 * 2 ^ 3

Using standard mathematical precedence rules, exponents are evaluated before multiplication. Thus, we'd solve this equation as: 5 * (2 ^ 3), which evaluates to 5 * 8 = 40.

Now, let's say we overloaded the XOR operator (^) to make it do exponents instead of bitwise XOR. According to C++ precedence rules, the multiplication operator takes precedence over the XOR operator. As a result, C++ would evaluate our mathematical equation above like this: (5 * 2) ^ 3. Consequently, it would produce the result 1000 instead of 40. This would be unintuitive to someone expecting the equation to be evaluated according to normal mathematical precedence rules. Thus, it would not be a good idea to overload the XOR operator in this way.

Unfortunately, there is no way to change the precedence of operators in C++.

mdg32871
March 30, 2009 at 11:44 pm · Reply

In the example above you were talking about overloading the XOR operator (^). Using this example to help me understand operator overloading, how would you go about overloading the ^? How would it be written? I realize that in this instance you wouldn't want to. I am just trying to understand this concept.

Alex
May 1, 2009 at 8:32 pm · Reply

The syntax for overloading operator^ is identical to the syntax for overloading operator+, which I cover in the next tutorial.

vkavicarm
February 18, 2013 at 2:09 pm · Reply

Hi Alex, how does the compiler differenciate as in if ^ is used for XOR or exponent?

Janez
March 15, 2015 at 12:27 am · Reply

Read the last paragraph in this tutorial for the answer.

Alex
March 29, 2016 at 8:51 am · Reply

^ is used for XOR, unless you've overridden it for one of your classes and redefined what it does (which I don't recommend doing due to the precedence issue).

Abhishek
January 14, 2008 at 5:07 am · Reply

compilers are very smart!

Alex
January 14, 2008 at 8:01 am · Reply

The people who write compilers are very smart! :)

ethicalstar
April 8, 2008 at 1:37 am · Reply

ofcourse! rightly said!!!

Darren
June 20, 2016 at 7:52 am · Reply

Though they could do with getting out more :P