21.1 — Introduction to operator overloading

learncpp.com/cpp-tutorial/introduction-to-operator-overloading/

In lesson 11.1 -- Introduction to 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 objects of a program-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 program-defined type, 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 program-defined types (e.g. one of your classes, or an enum type), the compiler will use the function overload resolution algorithm (described in lesson 11.3 -- Function overload resolution and ambiguous matches) to see if it can find an overloaded operator that is an unambiguous best match. This may involve implicitly converting one or more operands to match the parameter types of an overloaded operator. It may also involve implicitly converting program-defined types into fundamental types (via an overloaded typecast, which we'll cover later in this chapter) so that it can match a built-in operator. If no match can be found (or an ambiguous match is found), the compiler will 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 (.), pointer member 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 could overload `operator+(int, Mystring)`, but not `operator+(int, double)`.

Because standard library classes are considered to be user-defined, this means you could define `operator+(double, std::string)`. However, this is not a good idea because a future language standard could define this overload, which could break any programs that used your overload. Thus a best practice is that your overloaded operators should operate on at least one program-defined type. This guarantees that a future language standard won't potentially break your programs.

Best practice

An overloaded operator should operate on at least one program-defined type (either as a parameter of the function, or the implicit object).

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.

Best practice

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.

Best practice

If the meaning of an overloaded operator is not clear and intuitive, use a named function instead.

Finally, overloaded operators should return values in the way that is consistent with the original operators. Operators that do not modify their operands (e.g. arithmetic operators) should generally return results by value. Operators that modify their leftmost operand (e.g. pre-increment, any of the assignment operators) should generally return the leftmost operand by reference.

Best practice

Operators that do not modify their operands (e.g. arithmetic operators) should generally return results by value.

Operators that modify their leftmost operand (e.g. pre-increment, any of the assignment operators) should generally return the leftmost operand by reference.

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 program-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.