5.1 — Operator precedence and associativity

ALEX AUGUST 23, 2021

Chapter introduction

This chapter builds on top of the concepts from lesson 1.9 -- Introduction to literals and operators. A quick review follows:

In mathematics, an operation is a mathematical calculation involving zero or more input values (called operands) that produces a new value (called an output value). The specific operation to be performed is denoted by a construct (typically a symbol or pair of symbols) called an operator.

For example, as children we all learn that *2 + 3* equals *5*. In this case, the literals *2* and *3* are the operands, and the symbol *+* is the operator that tells us to apply mathematical addition on the operands to produce the new value *5*.

In this chapter, we'll discuss topics related to operators, and explore many of the common operators that C++ supports.

Operator precedence

Now, let's consider a more complicated expression, such as *4 + 2 * 3*. In order to evaluate this expression, we must understand both what the operators do, and the correct order to apply them. The order in which operators are evaluated in a compound expression is determined by an operator's precedence. Using normal mathematical precedence rules (which state that multiplication is resolved before addition), we know that the above expression should evaluate as *4 + (2 * 3)* to produce the value 10.

In C++, when the compiler encounters an expression, it must similarly analyze the expression and determine how it should be evaluated. To assist with this, all operators are assigned a level of precedence. Operators with the highest level of precedence are evaluated first.

You can see in the table below that multiplication and division (precedence level 5) have more precedence than addition and subtraction (precedence level 6). Thus, *4 + 2 * 3* evaluates as *4 + (2 * 3)* because multiplication has a higher level of precedence than addition.

Operator associativity

What happens if two operators in the same expression have the same precedence level? For example, in the expression *3 * 4 / 2*, the multiplication and division operators are both precedence level 5. In this case, the compiler can't rely upon precedence alone to determine how to evaluate the result.

If two operators with the same precedence level are adjacent to each other in an expression, the operator's associativity tells the compiler whether to evaluate the operators from left to right or from right to left. The operators in precedence level 5 have an associativity of left to right, so the expression is resolved from left to right: *(3 * 4) / 2 = 6*.

Table of operators

The below table is primarily meant to be a reference chart that you can refer back to in the future to resolve any precedence or associativity questions you have.

Notes:

- Precedence level 1 is the highest precedence level, and level 17 is the lowest. Operators with a higher precedence level get evaluated first.
- L->R means left to right associativity.
- R->L means right to left associativity.

Prec/Ass	Operator	Description	Pattern
1 None	::	Global scope (unary)	::name
	::	Namespace scope (binary)	class_name::member_name
2 L->R	()	Parentheses	(expression)
	()	Function call	function_name(parameters)
	()	Initialization	type name(expression)
	{}	Uniform initialization (C++11)	type name{expression}
	type()	Functional cast	new_type(expression)
	type{}	Functional cast (C++11)	new_type{expression}
	[]	Array subscript	pointer[expression]
	.	Member access from object	object.member_name
	->	Member access from object ptr	object_pointer->member_name
	++	Post-increment	lvalue++
	––	Post-decrement	lvalue––
	typeid	Run-time type information	typeid(type) or typeid(expression)
	const_cast	Cast away const	const_cast<type>(expression)
	dynamic_cast	Run-time type-checked cast	dynamic_cast<type>(expression)
	reinterpret_cast	Cast one type to another	reinterpret_cast<type>(expression)
	static_cast	Compile-time type-checked cast	static_cast<type>(expression)
	sizeof...	Get parameter pack size	sizeof...(expression)
	noexcept	Compile-time exception check	noexcept(expression)
	alignof	Get type alignment	alignof(Type)
3 R->L	+	Unary plus	+expression
	-	Unary minus	-expression
	++	Pre-increment	++lvalue
	––	Pre-decrement	––lvalue
	!	Logical NOT	!expression
	~	Bitwise NOT	~expression
	(type)	C-style cast	(new_type)expression
	sizeof	Size in bytes	sizeof(type) or sizeof(expression)
	co_await	Await asynchronous call	co_await expression
	&	Address of	&lvalue
	*	Dereference	*expression
	new	Dynamic memory allocation	new type
	new[]	Dynamic array allocation	new type[expression]
	delete	Dynamic memory deletion	delete pointer
	delete[]	Dynamic array deletion	delete[] pointer
4 L->R	->*	Member pointer selector	object_pointer->*pointer_to_member
	.*	Member object selector	object.*pointer_to_member
5 L->R	*	Multiplication	expression * expression
	/	Division	expression / expression
	%	Modulus	expression % expression
6 L->R	+	Addition	expression + expression
	•	Subtraction	expression - expression
7 L->R	<<	Bitwise shift left	expression << expression
	>>	Bitwise shift right	expression >> expression
8 L->R	<=>	Three-way comparison	expression <=> expression
9 L->R	<	Comparison less than	expression < expression
	<=	Comparison less than or equals	expression <= expression
	>	Comparison greater than	expression > expression

	>	Comparison greater than	expression > expression
	>=	Comparison greater than or equals	expression >= expression
10 L->R	==	Equality	expression == expression
	!=	Inequality	expression != expression
11 L->R	&	Bitwise AND	expression & expression
12 L->R	^	Bitwise XOR	expression ^ expression
13 L->R	|	Bitwise OR	expression | expression
14 L->R	&&	Logical AND	expression && expression
15 L->R	||	Logical OR	expression || expression
16 R->L	throw	Throw expression	throw expression
	co_yield	Yield expression	co_yield expression
	?:	Conditional	expression ? expression : expression
	=	Assignment	lvalue = expression
	*=	Multiplication assignment	lvalue *= expression
	/=	Division assignment	lvalue /= expression
	%=	Modulus assignment	lvalue %= expression
	+=	Addition assignment	lvalue += expression
	-=	Subtraction assignment	lvalue -= expression
	<<=	Bitwise shift left assignment	lvalue <<= expression
	>>=	Bitwise shift right assignment	lvalue >>= expression
	&=	Bitwise AND assignment	lvalue &= expression
	|=	Bitwise OR assignment	lvalue |= expression
	^=	Bitwise XOR assignment	lvalue ^= expression
17 L->R	,	Comma operator	expression, expression

You should already recognize a few of these operators, such as +, -, *, /, (), and sizeof. However, unless you have experience with another programming language, the majority of the operators in this table will probably be incomprehensible to you right now. That's expected at this point. We'll cover many of them in this chapter, and the rest will be introduced as there is a need for them.

> **Q: Where's the exponent operator?**
>
> C++ doesn't include an operator to do exponentiation (operator^ has a different function in C++). We discuss exponentiation more in lesson 5.3 -- Modulus and Exponentiation.

Parenthesization

In normal arithmetic, you learned that you can use parentheses to change the order of application of operations. For example, we know that *4 + 2 * 3* evaluates as *4 + (2 * 3)*, but if you want it to evaluate as *(4 + 2) * 3* instead, you can explicitly parenthesize the expression to make it evaluate the way you want. This works in C++ because parentheses have one of the highest precedence levels, so parentheses generally evaluate before whatever is inside them.

Now consider an expression like *x && y || z*. Does this evaluate as *(x && y) || z* or *x && (y || z)*? You could look up in the table and see that && takes precedence over ||. But there are so many operators and precedence levels that it's hard to remember them all. In order to reduce mistakes and make your code easier to understand without referencing a precedence table, it's a good idea to parenthesize any non-trivial compound expression, so it's clear what your intent is.

> **Best practice**
>
> Use parentheses to make it clear how an expression should evaluate, even if they are technically unnecessary.

The order of evaluation of expressions is mostly unspecified

Consider the following expression:

```
a + b
* c
```

We know from the precedence and associativity rules above that this expression will evaluate as if we had typed:

```
a + (b *
c)
```

If *a* is *1*, *b* is *2*, and *c* is 3, this expression will evaluate to the answer *7*.

However, the precedence and associativity rules only tell us how operators evaluate in relation to other operators. It does not tell us anything about the order in which the rest of the expression evaluates. For example, does variable *a*, *b*, or *c* get evaluated first?

Perhaps surprisingly, in many cases, the order of evaluation of any part of a compound expression (including function calls and argument evaluation) is unspecified. In such cases, the compiler free to choose any evaluation order it believes is optimal.

> **Warning**
>
> In many cases, the parts of a compound expression (including function calls and argument evaluation) may evaluate in any order.

For most expressions, this is irrelevant. In our sample expression above, it doesn't matter whether in which order variables *a*, *b*, or *c* are evaluated for their values: the answer will always be *7*. There is no ambiguity here.

But it is possible to write expressions where the order of evaluation does matter. Consider this program, which contains a mistake often made by new C++ programmers:

```cpp
#include <iostream>

int getValue()
{
    int x{};
    std::cin >> x;
    return x;
}

int main()
{
    std::cout << getValue() + (getValue() * getValue()); // a + (b * c)
    return 0;
}
```

If you run this program and enter inputs *1*, *2*, and *3*, you might assume that this program would print *7*. But that is making the assumption that the calls to getValue() will evaluate in left-to-right order. The compiler may choose a different order. For example, if the compiler chose a right-to-left order instead, the program would print *5* for the same set of inputs.

Best practice

Outside of the operator precedence and associativity rules, assume that the parts of an expression could evaluate in any order. Ensure that the expressions you write are not dependent on the order of evaluation of those parts.

Related content

There are some additional examples of cases where order of evaluation problems can occur in lesson 5.4 -- Increment/decrement operators, and side effects.

Quiz time

Question #1

You know from everyday mathematics that expressions inside of parentheses get evaluated first. For example, in the expression `(2 + 3) * 4`, the `(2 + 3)` part is evaluated first.

For this exercise, you are given a set of expressions that have no parentheses. Using the operator precedence and associativity rules in the table above, add parentheses to each expression to make it clear how the compiler will evaluate the expression.

Show Hint

Sample problem: x = 2 + 3 % 4
Binary operator % has higher precedence than operator + or operator =, so it gets evaluated first:

x = 2 + (3 % 4)

Binary operator + has a higher precedence than operator =, so it gets evaluated next:

Final answer: x = (2 + (3 % 4))

We now no longer need the table above to understand how this expression will evaluate.

a) x = 3 + 4 + 5;

Show Solution

b) x = y = z;

Show Solution

c) z *= ++y + 5;

Show Solution

d) a || b && c || d;

Show Solution

Next lesson
5.2 Arithmetic operators

Back to table of contents

Previous lesson
4.x Chapter 4 summary and quiz

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

191 COMMENTS

Newest