5.1 — Operator precedence and associativity

BY ALEX ON JUNE 13TH, 2007 | LAST MODIFIED BY ALEX ON AUGUST 21ST, 2019

Chapter introduction

This chapter builds on top of the concepts from lesson **1.8 -- 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) Namespace scope (binary)	::name class_name::member_name
2 L->R	() () () {} type() type{} [] . -> ++ –– typeid const_cast dynamic_cast reinterpret_cast static_cast	Parentheses Function call Initialization Uniform initialization (C++11) Functional cast Functional cast (C++11) Array subscript Member access from object Member access from object ptr Post-increment Post-decrement Run-time type information Cast away const Run-time type-checked cast Cast one type to another Compile-time type-checked cast	(expression) function_name(parameters) type name(expression) type name{expression} new_type(expression) new_type{expression} pointer[expression] object.member_name object_pointer->member_name lvalue++ lvalue–– typeid(type) or typeid(expression) const_cast<type>(expression) dynamic_cast<type>(expression) reinterpret_cast<type>(expression) static_cast<type>(expression)
3 R->L	+ - ++ –– ! ~ (type) sizeof & * new new[] delete delete[]	Unary plus Unary minus Pre-increment Pre-decrement Logical NOT Bitwise NOT C-style cast Size in bytes Address of Dereference Dynamic memory allocation Dynamic array allocation Dynamic memory deletion Dynamic array deletion	+expression -expression ++lvalue ––lvalue !expression ~expression (new_type)expression sizeof(type) or sizeof(expression) &lvalue *expression new type new type[expression] delete pointer delete[] pointer
4 L->R	->* .*	Member pointer selector Member object selector	object_pointer->*pointer_to_member object.*pointer_to_member
5 L->R	* / %	Multiplication Division Modulus	expression * expression expression / expression expression % expression
6 L->R	+ -	Addition Subtraction	expression + expression expression - expression
7 L->R	<< >>	Bitwise shift left Bitwise shift right	expression << expression expression >> expression
8 L->R	< <= > >=	Comparison less than Comparison less than or equals Comparison greater than Comparison greater than or equals	expression < expression expression <= expression expression > expression expression >= expression

9 L->R	== !=	Equality Inequality	expression == expression expression != expression
10 L->R	&	Bitwise AND	expression & expression
11 L->R	^	Bitwise XOR	expression ^ expression
12 L->R	|	Bitwise OR	expression | expression
13 L->R	&&	Logical AND	expression && expression
14 L->R	||	Logical OR	expression || expression
15 R->L	?: = *= /= %= += -= <<= >>= &= |= ^=	Conditional Assignment Multiplication assignment Division assignment Modulus assignment Addition assignment Subtraction assignment Bitwise shift left assignment Bitwise shift right assignment Bitwise AND assignment Bitwise OR assignment Bitwise XOR assignment	expression ? expression : expression lvalue = expression lvalue *= expression lvalue /= expression lvalue %= expression lvalue += expression lvalue -= expression lvalue <<= expression lvalue >>= expression lvalue &= expression lvalue |= expression lvalue ^= expression
16 R->L	throw	Throw expression	throw 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 parenthesis 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 parenthesis have one of the highest precedence levels, so parenthesis 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 parenthesis to make it clear how an expression should evaluate, even if they are technically unnecessary.

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


5.2 -- Arithmetic operators


Index


4.x -- Chapter 4 summary and quiz

C++ TUTORIAL | PRINT THIS POST

143 comments to 5.1 — Operator precedence and associativity

« Older Comments 1 2 3

tesfay
January 29, 2020 at 4:34 am · Reply

what about ambiguity in SLR,LR and operator precedence

« Older Comments 1 2 3