

6.1 — Operator precedence and associativity

by ALEX · JANUARY 17, 2024

Chapter introduction

This chapter builds on top of the concepts from lesson 1.9 -- Introduction to literals and operators (<https://www.learncpp.com/cpp-tutorial/introduction-to-literals-and-operators/>). A quick review follows:

An **operation** is a mathematical process 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 . Because there is only one operator being used here, this is straightforward.

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

Evaluation of compound expressions

Now, let's consider a compound expression, such as $4 + 2 * 3$. Should this be grouped as $(4 + 2) * 3$ which evaluates to 18 , or $4 + (2 * 3)$ which evaluates to 10 ? Using normal mathematical precedence rules (which state that multiplication is resolved before addition), we know that the above expression should be grouped as $4 + (2 * 3)$ to produce the value 10 . But how does the compiler know?

...



In order to evaluate an expression, the compiler must do two things:

- At compile time, the compiler must parse the expression and determine how operands are grouped with operators. This is done via the precedence and associativity rules, which we'll discuss momentarily.
- At compile time or runtime, the operands are evaluated and operations executed to produce a result.

Operator precedence

To assist with parsing a compound expression, all operators are assigned a level of precedence. Operators with a higher **precedence** level are grouped with operands first.

You can see in the table below that multiplication and division (precedence level 5) have a higher precedence level than addition and subtraction (precedence level 6). Thus, multiplication and division will be grouped with operands before addition and subtraction. In other words, $4 + 2 * 3$ will be grouped as $4 + (2 * 3)$.

Operator associativity

Consider a compound expression like $7 - 4 - 1$. Should this be grouped as $(7 - 4) - 1$ which evaluates to 2 , or $7 - (4 - 1)$, which evaluates to 4 ? Since both subtraction operators have the same precedence level, the compiler can not use precedence alone to determine how this should be grouped.

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. Subtraction has precedence level 6, and the operators in precedence

• • •



Table of operator precedence and associativity

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 have their operands grouped first.
- L->R means left to right associativity.
- R->L means right to left associativity.

Prec/Ass	Operator	Description	Pattern
1 L->R	::	Global scope (unary)	::name
	::	Namespace scope (binary)	class_name::member_name
2 L->R	()	Parentheses	(expression)
	()	Function call	function_name(arguments)
	type()	Functional cast	type(expression)
	type{}	List init temporary object (C++11)	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
	not	Logical NOT	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 (C++20)
	&	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
	%	Remainder	expression % expression
6 L->R	+	Addition	expression + expression
	-	Subtraction	expression - expression
7 L->R	<<	Bitwise shift left / Insertion	expression << expression
	>>	Bitwise shift right / Extraction	expression >> expression
8 L->R	<=	Three-way comparison (C++20)	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 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
	and	Logical AND	expression and expression
15 L->R		Logical OR	expression expression
	or	Logical OR	expression or expression
16 R->L	throw	Throw expression	throw expression
	co_yield	Yield expression (C++20)	co_yield expression
	?:	Conditional	expression ? expression : expression
	=	Assignment	lvalue = expression
	*=	Multiplication assignment	lvalue *= expression
	/=	Division assignment	lvalue /= expression
	%=	Remainder 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 is 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 [6.3 -- Remainder and Exponentiation](#) (<https://www.learnccpp.com/cpp-tutorial/remainder-and-exponentiation/>).

Note that `operator<<` handles both bitwise left shift and insertion, and `operator>>` handles both bitwise right shift and extraction. The compiler can determine which operation to perform based on the types of the operands.

Parenthesization

Due to the precedence rules, `4 + 2 * 3` will be grouped as `4 + (2 * 3)`. But what if we actually meant `(4 + 2) * 3`? Just like in normal mathematics, in C++ we can explicitly use parentheses to set the grouping of operands as we desire. This works because parentheses have one of the highest precedence levels, so parentheses generally evaluate before whatever is inside them.



Use parenthesis to make compound expressions easier to understand

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. And you don't want to have to look up operators all the time to understand how a compound expression evaluates.

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 a non-trivial compound expression should evaluate (even if they are technically unnecessary).

A good rule of thumb is: Parenthesize everything, except addition, subtraction, multiplication, and division.

There is one additional exception to the above best practice: Expressions that have a single assignment operator (and no comma operator) do not need to have the right operand of the assignment wrapped in parenthesis.



For example:

```
x = (y + z + w); // instead of this
x = y + z + w;    // it's okay to do this

x = ((y || z) && w); // instead of this
x = (y || z) && w;  // it's okay to do this

x = (y *= z); // expressions with multiple assignments still benefit from parenthesis
```

The assignment operators have the second lowest precedence (only the comma operator is lower, and it's rarely used). Therefore, so long as there is only one assignment (and no commas), we know the right operand will fully evaluate before the assignment.

Best practice

Expressions with a single assignment operator do not need to have the right operand of the assignment wrapped in parenthesis.

Value computation (of operations)

The C++ standard uses the term **value computation** to mean the execution of operators in an expression to produce a value. The precedence and association rules determine the order in which value computation happens.

For example, given the expression `4 + 2 * 3`, due to the precedence rules this groups as `4 + (2 * 3)`. `2 * 3` must be evaluated first, so that the resulting value of `6` can be used as the right operand of `operator+`.

Key insight

The precedence and associativity rules generally determine the order of value computation (of operators).

Order of evaluation (of operands)

The C++ standard (mostly) uses the term **evaluation** to refer to the evaluation of operands (not the evaluation of operators or expressions!). For example, given expression `a + b`, `a` will be evaluated to produce some value, and `b` will be evaluated to produce some value. These values can be then used as operands to `operator+` to compute a value.

• • •



Nomenclature

Informally, we typically use the term “evaluates” to mean the evaluation of an entire expression (value computation), not just the operands of an expression.

The order of evaluation of operands and function arguments is mostly unspecified [🔗](#) (#unspecified)

In most cases, the order of evaluation for operands and function arguments is unspecified, meaning they may be evaluated in any order.

Consider the following expression:

```
a * b + c * d
```

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

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

If `a` is 1, `b` is 2, `c` is 3, and `d` is 4, this expression will always compute the value 14.

However, the precedence and associativity rules only tell us how operators and operands are grouped and the order in which value computation will occur. They do not tell us the order in which the operands or subexpressions are evaluated. The compiler is free to evaluate operands `a`, `b`, `c`, or `d` in any order. The compiler is also free to calculate `a * b` or `c * d` first.

• • •



For most expressions, this is irrelevant. In our sample expression above, it doesn't matter whether in which order variables `a`, `b`, `c`, or `d` are evaluated for their values: the value calculated will always be 14. 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:

```

#include <iostream>

int getValue()
{
    std::cout << "Enter an integer: ";

    int x{};
    std::cin >> x;
    return x;
}

void printCalculation(int x, int y, int z)
{
    std::cout << x + (y * z);
}

int main()
{
    printCalculation(getValue(), getValue(), getValue()); // this line is ambiguous
    return 0;
}

```

If you run this program and enter the inputs 1, 2, and 3, you might assume that this program would calculate $1 + (2 * 3)$ and print 7. But that is making the assumption that the arguments to `printCalculation()` will evaluate in left-to-right order (so parameter `x` gets value 1, `y` gets value 2, and `z` gets value 3). If instead, the arguments evaluate in right-to-left order (so parameter `z` gets value 1, `y` gets value 2, and `x` gets value 3), then the program will print 5 instead.

Tip

The Clang compiler evaluates arguments in left-to-right order. The GCC compiler evaluates arguments in right-to-left order. You can run the above program on each of these compilers (e.g. on [Wandbox](https://wandbox.org/#) (<https://wandbox.org/#>)) and see for yourself.

The above program can be made unambiguous by making each function call to `getValue()` a separate statement:

```

#include <iostream>

int getValue()
{
    std::cout << "Enter an integer: ";

    int x{};
    std::cin >> x;
    return x;
}

void printCalculation(int x, int y, int z)
{
    std::cout << x + (y * z);
}

int main()
{
    int a{ getValue() }; // will execute first
    int b{ getValue() }; // will execute second
    int c{ getValue() }; // will execute third

    printCalculation(a, b, c); // this line is now unambiguous

    return 0;
}

```

In this version, `a` will always have value 1, `b` will have value 2, and `c` will have value 3. When the arguments to `printCalculation()` are evaluated, it doesn't matter which order the argument evaluation happens in -- parameter `x` will always get value 1, `y` will get value 2, and `z` will get value 3. This version will deterministically print 7.

Key insight

Operands, function arguments, and subexpressions may be evaluated in any order.

Warning

Ensure that the expressions (or function calls) you write are not dependent on operand (or argument) evaluation order.

Related content

Operators with side effects can also cause unexpected evaluation results. We cover this in lesson [6.4 -- Increment/decrement operators, and side effects](#) (<https://www.learncpp.com/cpp-tutorial/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 \(javascript:void\(0\)\)](#)

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 \(javascript:void\(0\)\)](#)

b) $x = y = z;$

[Show Solution \(javascript:void\(0\)\)](#)

c) $z *= ++y + 5;$

[Show Solution \(javascript:void\(0\)\)](#)

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

[Show Solution \(javascript:void\(0\)\)](#)



Next lesson

6.2 Arithmetic operators



[Back to table of contents](#)



Previous lesson

5.x Chapter 5 summary and quiz

Leave a comment...

 Name*

 Email* | ?

Notify me about replies: 

POST COMMENT

 Find a mistake? Leave a comment above! 

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

262 COMMENTS



Newest ▾

We and our partners share information on your use of this website to help improve your experience.



Do not sell my info: 

OKAY