6.1 — Operator precedence and associativity

Chapter introduction

This chapter builds on top of the concepts from lesson 1.9 -- 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 level 6 have an associativity of left to right. So this expression is grouped from left to right: `(7 - 4) - 1`.

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) Namespace scope (binary)	::name class_name::member_name

2 L->R	() () type() type{} [] . ->* ++ — typeid const_cast dynamic_cast reinterpret_cast static_cast sizeof… noexcept alignof	Parentheses Function call Functional cast List init temporary object (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 Get parameter pack size Compile-time exception check Get type alignment	(expression) function_name(arguments) type(expression) 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) sizeof…(expression) noexcept(expression) alignof(type)
3 R->L	+ - ++ — ! not ~ (type) sizeof co_await & * new new[] delete delete[]	Unary plus Unary minus Pre-increment Pre-decrement Logical NOT Logical NOT Bitwise NOT C-style cast Size in bytes Await asynchronous call Address of Dereference Dynamic memory allocation Dynamic array allocation Dynamic memory deletion Dynamic array deletion	+expression -expression ++lvalue —lvalue !expression not expression ~expression (new_type)expression sizeof(type) or sizeof(expression) co_await expression (C++20) &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 Remainder	expression * expression expression / expression expression % expression
6 L->R	+ -	Addition Subtraction	expression + expression expression - expression

7 L->R	<< >>	Bitwise shift left / Insertion Bitwise shift right / Extraction	expression << expression expression >> expression
8 L->R	<=>	Three-way comparison (C++20)	expression <=> expression
9 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
10 L->R	== !=	Equality Inequality	expression == expression 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	&& and	Logical AND Logical AND	expression && expression expression and expression
15 L->R	|| or	Logical OR Logical OR	expression || expression expression or expression
16 R->L	throw co_yield ?: = *= /= %= += -= <<= >>= &= |= ^=	Throw expression Yield expression (C++20) Conditional Assignment Multiplication assignment Division assignment Remainder assignment Addition assignment Subtraction assignment Bitwise shift left assignment Bitwise shift right assignment Bitwise AND assignment Bitwise OR assignment Bitwise XOR assignment	throw expression co_yield expression 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
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.

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

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) 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.

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