3.1 — Operator precedence and associativity

BY ALEX ON JUNE 13TH, 2007 | LAST MODIFIED BY ALEX ON JUNE 1ST, 2018

In mathematics, an **operation** is a mathematical calculation involving zero or more input values (called **operands**) that produces an output value. Common operations (such as addition) use special symbols (such as +) that denote the operation. These symbols are called **operators**. Operators in programming work the same way except the names may not always be a symbol. Operators work analogously to functions that take input parameters and return a value, except they are more concise. For example, 4 + 2 * 3 is much easier to read than add(4, mult(2, 3))!

In order to properly evaluate an expression such as `4 + 2 * 3`, 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 called **operator 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. Those with the highest precedence are evaluated first. You can see in the table below that multiplication and division (precedence level 5) have a higher precedence than addition and subtraction (precedence level 6). The compiler uses these levels to determine how to evaluate expressions it encounters.

Thus, 4 + 2 * 3 evaluates as 4 + (2 * 3) because multiplication has a higher level of precedence than addition.

If two operators with the same precedence level are adjacent to each other in an expression, the **operator associativity** rules tell the compiler whether to evaluate the operators from left to right or from right to left. For example, in the expression `3 * 4 / 2`, the multiplication and division operators are both precedence level 5. Level 5 has an associativity of left to right, so the expression is resolved from left to right: `(3 * 4) / 2 = 6`.

Table of operators

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
	::	Class 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)
3 R->L	+	Unary plus	+expression
	-	Unary minus	-expression
	++	Pre-increment	++lvalue

	— ! ~ (type) sizeof & * new new[] delete delete[]	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	—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 (see note below) 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

Note: The expression in the middle of the conditional operator ?: is evaluated as if it were parenthesized.

A few operators you should already recognize: +, -, *, /, (), =, <, >, <=, and >=. These arithmetic and relational operators have the same meaning in C++ as they do in every-day usage.

However, unless you have experience with another programming language, it’s likely the majority of the operators in this table will 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.

The above 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.

That said, if you have an expression that uses operators of different types, it is a best practice to use parenthesis to explicitly disambiguate the order of evaluation.

Rule: If your expression uses different operators, use parenthesis to make it clear how the expression should evaluate, even if they are technically unnecessary.

How do I do exponents?

You'll note that the ^ operator (commonly used to denote exponentiation in standard mathematical nomenclature) is a Bitwise XOR operation in C++. C++ does not include an exponent operator. To do exponents in C++, #include the <cmath> header, and use the pow() function:

```
#include <cmath>

double x = std::pow(3.0, 4.0); // 3 to the 4th power
```

Note that the parameters (and return value) of pow in the above example are of type double. std::pow() assumes the base is a floating point number. Note that due to rounding errors in floating point numbers, the results of pow() may not be precise (slightly smaller or larger than what you'd expect).

If you want to do integer base, you're best off just using your own function to do so, like this one (that uses the "exponentiation by squaring" algorithm for efficiency):

```
// note: exp must be non-negative
int pow(int base, int exp)
{
    int result = 1;
    while (exp)
    {
        if (exp & 1)
            result *= base;
        exp >>= 1;
        base *= base;
    }

    return result;
}
```

Don't worry if you don't understand all of the parts of this function yet. Just beware of overflowing your integer result, which can happen very quickly if either argument is large.

Quiz

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.

Hint: Use the pattern column in the table above to determine whether the operator is unary (has one operand) or binary (has two operands). Review section **1.5 -- A first look at operators** if you need a refresher on what unary and binary operators are.

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;
b) x = y = z;
c) z *= ++y + 5;
d) a || b && c || d;

Solutions

1) **Hide Solution**

a) Binary operator + has higher precedence than =:

x = (3 + 4 + 5);

Binary operator + has left to right association:

Final answer: x = ((3 + 4) + 5);

b) Binary operator = has right to left association:

Final answer: x = (y = z);

c) Unary operator ++ has the highest precedence:

z *= (++y) + 5;

Binary operator + has the next highest precedence:

Final answer: z *= ((++y) + 5);

d) Binary operator && has higher precedence than ||:

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

Binary operator || has left to right association:

Final answer: (a || (b && c)) || d;

3.2 -- Arithmetic operators

Index

2.10 -- Chapter 2 comprehensive quiz

114 comments to 3.1 — Operator precedence and associativity

« Older Comments 1 2