Order of evaluation

Order of evaluation of the operands of almost all C++ operators (including the order of evaluation of function arguments in a function-call expression and the order of evaluation of the subexpressions within any expression) is unspecified. The compiler can evaluate operands in any order, and may choose another order when the same expression is evaluated again.

There are exceptions to this rule which are noted below.

Except where noted below, there is no concept of left-to-right or right-to-left evaluation in C++. This is not to be confused with left-to-right and right-to-left associativity of operators: the expression `f1() + f2() + f3()` is parsed as `(f1() + f2()) + f3()` due to left-to-right associativity of operator+, but the function call to `f3` may be evaluated first, last, or between `f1()` or `f2()` at run time.

Sequenced-before rules (since C++11)

Definitions

Evaluations

There are two kinds of evaluations performed by the compiler for each expression or subexpression (both of which are optional):

- *value computation*: calculation of the value that is returned by the expression. This may involve determination of the identity of the object (glvalue evaluation, e.g. if the expression returns a reference to some object) or reading the value previously assigned to an object (prvalue evaluation, e.g. if the expression returns a number, or some other value)

- *side effect*: access (read or write) to an object designated by a volatile glvalue, modification (writing) to an object, calling a library I/O function, or calling a function that does any of those operations.

Ordering

"sequenced-before" is an asymmetric, transitive, pair-wise relationship between evaluations within the same thread.

- If A is sequenced before B, then evaluation of A will be complete before evaluation of B begins.

- If A is not sequenced before B and B is sequenced before A, then evaluation of B will be complete before evaluation of A begins.

- If A is not sequenced before B and B is not sequenced before A, then two possibilities exist:

  - evaluations of A and B are unsequenced: they may be performed in any order and may overlap (within a single thread of execution, the compiler may interleave the CPU instructions that comprise A and B)

  - evaluations of A and B are indeterminately sequenced: they may be performed in any order but may not overlap: either A will be complete before B, or B will be complete before A. The order may be the opposite the next time the same expression is evaluated.

Rules

1) Each value computation and side effect of a *full expression*, that is

- unevaluated operand

- constant expression

- an entire initializer, including any comma-separated constituent expressions

- the destructor call generated at the end of the lifetime of a non-temporary object

- an expression that is not part of another full-expression (such as the entire expression statement, controlling expression of a for/while loop, conditional expression of if/switch, the expression in a return statement, etc),

including implicit conversions applied to the result of the expression, destructor calls to the temporaries, default member initializers (when initializing aggregates), and every other language construct that involves a function call, is *sequenced before* each value computation and side effect of the next *full expression*.

2) The value computations (but not the side-effects) of the operands to any operator are *sequenced before* the value computation of the result of the operator (but not its side-effects).

3) When calling a function (whether or not the function is inline, and whether or not explicit function call syntax is used), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is *sequenced before* execution of every expression or statement in the body of the called function.

4) The value computation of the built-in post-increment and post-decrement operators is *sequenced before* its side-effect.

5) The side effect of the built-in pre-increment and pre-decrement operators is *sequenced before* its value computation (implicit rule due to definition as compound assignment)

6) Every value computation and side effect of the first (left) argument of the built-in logical AND operator `&&` and the built-in logical OR operator `||` is sequenced before every value computation and side effect of the second (right) argument.
7) Every value computation and side effect associated with the first expression in the conditional operator `?:` is *sequenced before* every value computation and side effect associated with the second or third expression.
8) The side effect (modification of the left argument) of the built-in assignment operator and of all built-in compound assignment operators is *sequenced after* the value computation (but not the side effects) of both left and right arguments, and is *sequenced before* the value computation of the assignment expression (that is, before returning the reference to the modified object)
9) Every value computation and side effect of the first (left) argument of the built-in comma operator `,` is *sequenced before* every value computation and side effect of the second (right) argument.
10) In list-initialization, every value computation and side effect of a given initializer clause is *sequenced before* every value computation and side effect associated with any initializer clause that follows it in the brace-enclosed comma-separated list of initalizers.
11) A function call that is not *sequenced before* or *sequenced after* another function call is *indeterminately sequenced* (the program must behave as if the CPU instructions that constitute different function calls were not interleaved, even if the functions were inlined).

The rule 11 has one exception: a function calls made by a standard library algorithm executing under `std::par_unseq` execution policy are unsequenced and may be arbitrarily interleaved. (since C++17)

12) The call to the allocation function (`operator new`) is indeterminately sequenced with respect to (until C++17)sequenced before (since C++17) the evaluation of the constructor arguments in a new-expression

When returning from a function, copy-initialization of the temporary that is the result of evaluating the function call is *sequenced-before* the destruction of all temporaries at the end of the operand of the return statement, which, in turn, is *sequenced-before* the destruction of local variables of the block enclosing the return statement. (since C++14)

In a function-call expression, the expression that names the function is sequenced before every argument expression and every default argument. (since C++17)

In a function call, value computations and side effects of the initialization of every parameter are indeterminately sequenced with respect to value computations and side effects of any other parameter.

Every overloaded operator obeys the sequencing rules of the built-in operator it overloads when called using operator notation.

In a subscript expression `E1[E2]`, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2

In a pointer-to-member expression `E1.*E2` or `E1->*E2`, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2 (unless the dynamic type of E1 does not contain the member to which E2 refers)

In a shift operator expression `E1<<E2` and `E1>>E2`, every value computation and side-effect of E1 is sequenced before every value computation and side effect of E2

In every simple assignment expression `E1=E2` and every compound assignment expression `E1@=E2`, every value computation and side-effect of E2 is sequenced before every value computation and side effect of E1

Every expression in a comma-separated list of expressions in a parenthesized initializer is evaluated as if for a function call (indeterminately-sequenced)

Undefined behavior

1) If a side effect on a scalar object is unsequenced relative to another side effect on the same scalar object, the behavior is undefined.

```
i = ++i + 2;       // undefined behavior until C++11
i = i++ + 2;       // undefined behavior until C++17
f(i = -2, i = -2); // undefined behavior until C++17
f(++i, ++i);       // undefined behavior until C++17, unspecified after C+
+17
i = ++i + i++;     // undefined behavior
```

2) If a side effect on a scalar object is unsequenced relative to a value computation using the value of the same scalar object, the behavior is undefined.

```
cout << i << i++; // undefined behavior until C++17
a[i] = i++;       // undefined behavior until C++17
n = ++i + i;      // undefined behavior
```

Sequence point rules **(until C++11)**

Definitions

Evaluation of an expression might produce side effects, which are: accessing an object designated by a volatile lvalue, modifying an object, calling a library I/O function, or calling a function that does any of those operations.

A *sequence point* is a point in the execution sequence where all side effects from the previous evaluations in the sequence are complete, and no side effects of the subsequent evaluations started.

Rules

1) There is a sequence point at the end of each full expression (typically, at the semicolon).

2) When calling a function (whether or not the function is inline and whether or not function call syntax was used), there is a sequence point after the evaluation of all function arguments (if any) which takes place before execution of any expressions or statements in the function body.

3) There is a sequence point after the copying of a returned value of a function and before the execution of any expressions outside the function.

4) Once the execution of a function begins, no expressions from the calling function are evaluated until execution of the called function has completed (functions cannot be interleaved).

5) In the evaluation of each of the following four expressions, using the built-in (non-overloaded) operators, there is a sequence point after the evaluation of the expression `a`.

```
a && b
a || b
a ? b : c
a , b
```

Undefined behavior

1) Between the previous and next sequence point a scalar object must have its stored value modified at most once by the evaluation of an expression, otherwise the behavior is undefined.

```
i = ++i + i++; // undefined behavior
i = i++ + 1; // undefined behavior
i = ++i + 1; // undefined behavior (well-defined in C++11)
++ ++i; // undefined behavior (well-defined in C++11)
f(++i, ++i); // undefined behavior
f(i = -1, i = -1); // undefined behavior
```

2) Between the previous and next sequence point, the prior value of a scalar object that is modified by the evaluation of the expression, must be accessed only to determine the value to be stored. If it is accessed in any other way, the behavior is undefined.

```
cout << i << i++; // undefined behavior
a[i] = i++; // undefined behavior
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 1885	C++14	sequencing of the destruction of automatic variables on function return was not explicit	sequencing rules added

See also

- Operator precedence which defines how expressions are built from their source code representation.
- When parsing an expression, an operator which is listed on some row of the table above with a precedence will be bound tighter (as if by parentheses) to its arguments

than any operator that is listed on a row further below it with a lower precedence. For example, the expressions `std::cout << a & b` and `*p++` are parsed as `(std::cout << a) & b` and `*(p++)`, and not as `std::cout << (a & b)` or `(*p)++`.

- Operators that have the same precedence are bound to their arguments in the direction of their associativity. For example, the expression `a = b = c` is parsed as `a = (b = c)`, and not as `(a = b) = c` because of right-to-left associativity of assignment, but `a + b - c` is parsed `(a + b) - c` and not `a + (b - c)` because of left-to-right associativity of addition and subtraction.
- Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`delete ++*p` is `delete(++(*p))`) and unary postfix operators always associate left-to-right (`a[1][2]++` is `((a[1])[2])++`). Note that the associativity is meaningful for member access operators, even though they are grouped with unary postfix operators: `a.b++` is parsed `(a.b)++` and not `a.(b++)`.
- Operator precedence is unaffected by operator overloading. For example, `std::cout << a ? b : c;` parses as `(std::cout << a) ? b : c;` because the precedence of arithmetic left shift is higher than the conditional operator.

- ## [] Notes

- Precedence and associativity are compile-time concepts and are independent from order of evaluation, which is a runtime concept.

- The standard itself doesn't specify precedence levels. They are derived from the grammar.

- `const_cast`, `static_cast`, `dynamic_cast`, `reinterpret_cast`, `typeid`, `sizeof...`, `noexcept` and `alignof` are not included since they are never ambiguous.

- Some of the operators have alternate spellings (e.g., `and` for `&&`, `or` for `||`, `not` for `!`, etc.).
- Relative precedence of the ternary conditional and assignment operators differs between C and C++: in C, assignment is not allowed on the right-hand side of a ternary conditional operator, so `e = a < d ? a++ : a = d` cannot be parsed. Many C compilers use a modified grammar where `?:` has higher precedence than `=`, which parses that as `e = ( ((a < d) ? (a++) : a) = d )` (which then fails to compile because `?:` is never lvalue in C and `=` requires lvalue on the left). In C++, `?:` and `=` have equal precedence and group right-to-left, so that `e = a < d ? a++ : a = d` parses as `e = ((a < d) ? (a++) : (a = d))`.