Refining Expression Evaluation Order for Idiomatic C++

Gabriel Dos Reis Herb Sutter Jonathan Caves

Abstract

This paper proposes an order of evaluation of operands in expressions, directly supporting decades-old established and recommended C++ idioms. The result is the removal of embarrassing traps for novices and experts alike, increased confidence and safety of popular programming practices and facilities, hallmarks of modern C++.

1. INTRODUCTION

Order of expression evaluation is a recurring discussion topic in the C++ community. In a nutshell, given an expression such as **f(a, b, c)**, the order in which the sub-expressions **f**, **a**, **b**, **c** (which are of arbitrary shapes) are evaluated is left *unspecified* by the standard. If any two of these sub-expressions happen to modify the same object without intervening sequence points, the behavior of the program is undefined. For instance, the expression **f(i++, i**) where **i** is an integer variable leads to undefined behavior, as does **v[i] = i++**. Even when the behavior is not undefined, the result of evaluating an expression can still be anybody's guess. Consider the following program fragment:

```
#include <map>
int main() {
    std::map<int, int> m;
    m[0] = m.size();                // #1
}
```

What should the map object **m** look like after evaluation of the statement marked #1? **{{0, 0}}** or **{{0, 1}}**?

1.1. CHANGES FROM PREVIOUS VERSIONS

a. The original version of this proposal (Dos Reis, et al., 2014) received unanimous support from the Evolution Working Group (EWG) at the Fall 2014 meeting in Urbana, IL, as approved direction, and also strong support for inclusion in C++17. The most fundamental delta in this revision, compared to that document, is the inclusion of formal wording for approval into the Working Draft.
b. Additionally, EWG suggested inclusion of a few more operators.

c. We added a couple of sections expanding the rationale behind proposed changes.
d. At the Fall 2015 meeting in Kona, HI, during review by the Core Working Group, some members of the Core Working Group suggested a variation of the evaluation of function calls as a, separate, subsidiary proposal. This revision includes two variations for that rule (see section 8.)
e. Explicit use of the phrasing "*value computation and side effects associated with*" [CWG suggestion at the Fall 2015 meeting in Kona, HI]
f. Introduced a new "text definition" for "expression X sequenced before expression Y" to replace the phrase introduced in (e.) [CWG suggestion at the Spring 2016 meeting in Jacksonville, FL]

2. A CORRODING PROBLEM

These questions aren't for entertainment, or job interview drills, or just for academic interests. The order of expression evaluation, as it is currently specified in the standard, undermines advices, popular programming idioms, or the relative safety of standard library facilities. The traps aren't just for novices or the careless programmer. They affect all of us indiscriminately, even when we know the rules.

Consider the following program fragment:

```
void f()
{
   std::string s = "but I have heard it works even if you don't believe in it";
   s.replace(0, 4, "").replace(s.find("even"), 4, "only").replace(s.find(" don't"), 6, "");
   assert(s == "I have heard it works only if you believe in it");
}
```

The assertion is supposed to validate the programmer's intended result. It uses "chaining" of member function calls, a common standard practice. This code has been reviewed by C++ experts world-wide, and published (The C++ Programming Language, 4th edition.) Yet, its vulnerability to unspecified order of evaluation has been discovered only recently by a tool. Even if you would like to blame the "excessive" chaining, remember that expressions of the form **std::cout << f() << g() << h()** usually result in chaining, after the overloaded operators have been resolved into function calls. It is the source of endless headaches. Newer library facilities such as **std::future<T>** are also vulnerable to this problem, when considering chaining of the **then()** member function to specify a sequence of computation. The solution isn't to avoid chaining. Rather, it is to fix the problem at the source: refinement of the language rules.

3. WHY NOW?

The current rules have been in effect for more than three decades. So, why change them now? Well, a programming language is a set of responses to challenges of its time. Many of the existing rules regarding order of expression evaluation made sense when C was designed and in the constrained environment

where C++ was originally designed and implemented. Some of the justifications probably still hold today. However, a living and evolving programming language cannot just hold onto inertia.

The language should support contemporary idioms. For example, using **<<** as insertion operator into a stream is now an elementary idiom. So is chaining member function calls. The language rules should guarantee that such idioms aren't programming hazards. We have library facilities (e.g. **std::future<T>**) designed to be used idiomatically with chaining. Without the guarantee that the obvious order of evaluation for function call and member selection is obeyed, these facilities become traps, source of obscure, hard to track bugs, facile opportunities for vulnerabilities.

The language should support our programming. The changes suggested below are conservative, pragmatic, with one overriding guiding principle: ***effective support for idiomatic C++***. In particular, when choosing between several alternatives, we look for what will provide better support for existing idioms, what will nurture and sustain new programming techniques. Considerations such as how an expression is internally elaborated (e.g. function call), while important, are secondary. The primary focus is on what the programmer reads and writes, in particular in generic codes, not what the compiler internally does according to fairly arcane rules. By generic codes, we don't just mean "template codes". We do also consider "normal" application codes using common notations for conceptually same operations. For example, consider the expression **ary[idx] = expr**, a rule that applies uniformly whether **ary** is a built-in (dense) array or an associative (sparse) array increases the set of types that **ary** can take on, hence supports generic programming. Observe that operators are generally preferred in C++ generic codes because they cover larger surface than member functions calls, although versions of the uniform function call syntax may alleviate that to some extent. Even with uniform function call syntax, we still do not know, looking at a generic code fragment, whether a particular operator or function will resolve to a member function or not; consequently, the low-level mechanics (which happen after instantiation) should, ideally, not be the driving force of the choice. Rather, the driver seat should be given to idioms.

4. A SOLUTION

We propose to revise C++ evaluation rules to support decades-old idiomatic constructs and programming practices. A simple solution would be to require that every expression has a well-defined evaluation order. That suggestion has traditionally met resistance for various reasons. Rather, this proposal suggests a more targeted fix:

- Postfix expressions are evaluated from left to right. This includes functions calls and member selection expressions.
- Assignment expressions are evaluated from right to left. This includes compound assignments.
- Operands to shift operators are evaluated from left to right.

In summary, the following expressions are evaluated in the order **a**, then **b**, then **c**, then **d**:

1. **a.b**
2. **a->b**

3. **a->*b**
4. **a(b1, b2, b3)**
5. **b @= a**
6. **a[b]**
7. **a << b**
8. **a >> b**

Furthermore, we suggest the following additional rule: **the order of evaluation of an expression involving an overloaded operator is determined by the order associated with the corresponding built-in operator, not the rules for function calls**. This rule is to support generic programming and extensive use of overloaded operators, which are distinctive features of modern C++.

A second, subsidiary proposal replaces the evaluation order of function calls as follows: the function is evaluated before all its arguments, but any pair of arguments (from the argument list) is indeterminately sequenced; meaning that one is evaluated before the other but the order is not specified; it is guaranteed that the function is evaluated before the arguments. This reflects a suggestion made by some members of the Core Working Group.

5. Postfix Increment and Decrement

At the Fall 2014 meeting in Urbana, IL, Clark Nelson observed that the proposal does not suggest when side effects of postfix increment and postfix decrement are "committed". Indeed, the current proposal does not suggest any particular modification to the sequencing of unary expressions. The primary reason is that we have not found a choice that will support an existing widely used programming idiom or nurture new programming techniques. Consequently, at this point, we do not propose any change to unary expressions. The side effects of unary expressions shall be committed before the next expression (if any) is evaluated if it is part of a binary expression or a function call. The sequencing order of unary expressions is not changed by this proposal.

6. Formal Wording

The following changes are against N4527, the current Working Draft.

6.1. General (Clause 1)

- Add to paragraph 1.9/13

  An expression *X* is said to be sequenced before an expression *Y* if every value computation and every side effect associated with the expression *X* is sequenced before every value computation and every side effect associated with the expression *Y*.

- Remove note from 1.9/16

  ~~[*Note:* Value computations and side effects associated with different argument expressions are unsequenced. — *end note*]~~

6.2. EXPRESSIONS (CLAUSE 5)

- Change paragraph 5/2 as follows:

  [*Note:* Operators can be overloaded, that is, given meaning when applied to expressions of class type (Clause 9) or enumeration type (7.2). Uses of overloaded operators are transformed into function calls as described in 13.5. Overloaded operators obey the rules for syntax and evaluation order specified in Clause 5, but the requirements of operand type~~,~~ and value category~~, and evaluation order~~ are replaced by the rules for function call. Relations between operators, such as ++**a** meaning **a**+=**1**, are not guaranteed for overloaded operators (13.5), and are not guaranteed for operands of type **boo**l. —*end note*]

- Add to paragraph 5.2.1/1:

  except that in the case of an array operand, the result is an lvalue if that operand is an lvalue and an xvalue otherwise.The expression E1 is sequenced before the expression E2.

- Modify paragraph 5.2.2/4:

  When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument. ~~[*Note:* Such initializations are indeterminately sequenced with respect to each other — *end note*]~~ If the function is a non-static member function, the this parameter of the function (9.3.2) shall be initialized with a pointer to the object of the call, converted as if by an explicit type conversion (5.4). The *postfix-expression* is sequenced before each expression in the *expression-list* and any default argument. Every value computation and side effect associated with the initialization of a parameter, and the initialization itself, is sequenced before every value computation and side effect associated with the initialization of any subsequent parameter.
  [Example:

  ```
  void f()
  {
    std::string s = “but I have heard it works even if you don’t believe in it”;
    s.replace(0, 4, “”).replace(s.find(“even”), 4, “only”).replace(s.find(“ don’t”), 6, “”);
    assert(s == “I have heard it works only if you believe in it”); // OK.
  }
  ```

  --end example]

- Remove paragraph 5.2.2/8.
  ~~[*Note:* The evaluations of the postfix expression and of the arguments are all unsequenced relative to one another. All side effects of argument evaluations are sequenced before the function is entered (see 1.9). —*end note*]~~

- Modify paragraph 5.3.4/18 as follows
  The invocation of the allocation function is ~~indeterminately~~ sequenced ~~with respect to~~ before the evaluations of the expressions in the *new-initializer*. Initialization of the allocated object is sequenced before the value computation in the *new-expression*. ~~It is unspecified whether expressions in the new-initializer are evaluated if the allocation function returns the null pointer or exits using an exception.~~

- Remove this phrase from 5.3.4/10
  If any part of the object initialization described above78 terminates by throwing an exception~~, storage has been obtained for the object,~~ and a suitable deallocation function can be found, the deallocation function is called to free the memory in which the object was being constructed, after which the exception continues to propagate in the context of the *new-expression*.

- Append to paragraph 5.5/4

  If the dynamic type of E1 does not contain the member to which E2 refers, the behavior is undefined. Otherwise, the expression E1 is sequenced before the expression E2.

- Add a new paragraph 5.8/4 to section 5.8

  The expression E1 is sequenced before the expression E2.

- Add to paragraph 5.18/1
  In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression. The right operand is sequenced before the left operand.
  .

- Remove footnote 88 from paragraph 5.19/1.

~~88) However, an invocation of an overloaded comma operator is an ordinary function call; hence, the evaluations of its argument expressions are unsequenced relative to one another (see 1.9).~~

6.3. EXPRESSION LIST IN INITIALIZERS (SECTION 8.5)

Add a new paragraph 8.5/19

If the initializer is a parenthesized *expression-list*, the expressions are evaluated in the order specified for function calls (5.2.2).

6.4. OVERLOADED OPERATORS (CLAUSE 13)

- Append to paragraph 13.3.1.2/2

Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 10(where @ denotes one of the operators covered in the specified subclause). However, the operands are sequenced in the order prescribed for the built-in operator (Clause 5).

7. IMPLEMENTATION EXPERIENCE REPORT

We modified the Visual C++ compiler to measure the impacts in the worst case scenario. That is

- Keep the **existing optimizers as they are with no new optimization that exploits the proposed evaluation rules** (so that the optimizers are artificially 'hampered')
- **Forcefully impose a left-to-right evaluation of argument list in function calls** (except the in case where there is a documented existing bug being separately addressed)

We successfully built, installed, and booted the NT kernel. Then we built a large application code base, and ran "build, validation, test suites." That uncovered sources of potential bugs due to non-portability assumptions: one real-world-code test failed, out of 26. Then, we compiled and ran Spec benchmarks. We found that some entries in the benchmark suite ran slower, others ran faster compared to the scenario

where the evaluation of the argument list is left unspecified. The variation is between -4% and +4%. **It is worth noting that these results are for the worst case scenario where the optimizers have not yet been updated to be aware of, and take advantage of the new evaluation rules and they are blindly forced to evaluate function calls from left to right**. It is clear that the left-to-right evaluation strategy is triggering new optimization paths (different inlining decisions and different register allocation) affecting the variations in the benchmark performance. If appears those opportunities have not traditionally been exploited, even though permitted under the unspecified order regime.

Based on these experiments, we feel confident recommending the left-to-right evaluation rules for syntactic function calls and in the functional cast notation involving more than one arguments in the argument list.

8. Alternate Evaluation Order for Function Calls

During the wording review at the Fall 2015 meeting in Kona, HI, some members of CWG expressed a desire for an alternate evaluation rule for function calls: the expression in the function position is evaluated before all the arguments and the evaluations of the arguments are indeterminately sequenced, but with no interleaving. We do not believe that such a nondeterminism brings any substantial added optimization benefit, but it does perpetuate the confusion and hazards around order of evaluations in function calls. It perpetuates unnecessary confusion around brace-initialization vs. direct initialization using parenthesis. Such an ordering would be implemented by the following requirement added to 5.2.2/4:

> The value computation and associated side-effect of the postfix-expression are sequenced before those of the expressions in the *expression-list*. The initializations of the declared parameters are indeterminately sequenced with no interleaving.

At the Spring 2016 meeting in Jacksonville, FL, EWG decided (via a poll) not to pursue this alternative evaluation order.

9. Acknowledgement

Thanks to Bjarne Stroustrup for discussing this issue with us. We acknowledge the numerous people who contributed to the discussions on the committee reflectors, as well as in private, including Chris Hawblitzel, Jim Hogg, Jason Merrill, Gor Nishanov, Andrew Pardoe, Dave Sielaff, and Jim Springfield. CWG provided useful feedback for presentation and helped with more accurate reflection of the design in the formal wording.

10. References

Gabriel Dos Reis, Herb Sutter and Jonathan Caves Refining Expression Evaluation Order for Idiomatic C++ [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4228.pdf]. - 2014. - Doc. N4228.