Refining Expression Evaluation Order for Idiomatic C++ (Revision 1)

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

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. Additionally, EWG suggested inclusion of a few more operators. Finally, we added a couple of sections expanding the rationale behind proposed changes.

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 than 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 recent proposals will 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, c, d)**
4. **b @= a**
5. **{ a, b, c, d }**
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++.

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. 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:

  The expression E1 is sequenced before 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~~ sequenced from left to right (1.9) —*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). This initialization is sequenced before the initialization of the function parameters, if any. *The postfix-expression* designating the function is sequenced before the initialization of any parameters, including **this**.

- Add to paragraph 5.5/4

  Otherwise, the expression E1 is sequenced before E2.

- Add a new paragraph 5.8/4 to section 5.8

  In both left-shifted and right-shifted expressions, the operand E1 is sequenced before E2.

- Add to paragraph 5.18/2

  The operands are sequenced from right to left.

- Modify paragraph 5.18/7:

  The behavior of an expression of the form **E1 *op* = E2** is equivalent to **E1 = E1 *op* E2** except that **E1** is evaluated only once and **E2** is sequenced before **E1**. In **+=** and **-=**, **E1** shall either have arithmetic type or be a pointer to a possibly cv-qualified completely-defined object type. In all other cases, **E1** shall have arithmetic type.

6.2. OVERLOADED OPERATORS (CLAUSE 13)

- Add to paragraph 13.3.1.2/2

  However, the operands are sequenced in the order prescribed for the built-in operators (Clause 5).

1 ACKNOWLEDGEMENT

Thanks to Bjarne Stroustrup for discussing this issue with us. Eric Brumer provided experimental implementation of these rules for evaluation. We acknowledge the numerous people who contributed to the discussions on the committee reflectors, as well as in private, including Chris Hawblitzel, Jim Hogg, Gor Nishanov, Dave Sielaff, and Jim Springfield.

2 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.