Document: N4228
Date: 2014-10-13
Working Group: EWG
Reply-To: gdr@microsoft.com

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

Expression evaluation order 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** 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 guest. 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}}**?

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 << a << b << c** 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, 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.

3. 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 proposes suggests a more targeted fix:

- Postfix expressions are evaluated from left to right. This includes functions calls and member section 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**:

- **a.b**
- **a->b**
- **a(b, c, d)**
- **b = a**
- **b @= a**

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**

operators, not the rules for function calls. This rule is to support generic programming and extensive use of overloaded operators, distinctive features of modern C++.

4. Formal Wording

TBD.

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.