What is the order of evaluation in C#?

devblogs.microsoft.com/oldnewthing/20070814-00

August 14, 2007

Raymond Chen

The C and C++ languages leave the order of evaluation generally unspecified aside from specific locations called *sequence points*. Side effects of operations performed prior to the sequence point are guaranteed visible to operations performed after it.¹ For example, the C comma operator introduces a sequence point. When you write `f(), g()`, the language guarantees that any changes to program state made by the function `f` can be seen by the function `g`; `f` executes before `g`. On the other hand, the multiplication operator does not introduce a sequence point. If you write `f() * g()` there is no guarantee which side will be evaluated first.

(Note that order of evaluation is not the same as associativity and operator precedence. Given the expression `f() + g() * h()`, operator precedence says that it should be evaluated as if it were written `f() + (g() * h())`, but that doesn't say what order the three functions will be evaluated. It merely describes how the results of the three functions will be combined.)

In the C# language, the order of evaluation is spelled out more explicitly. The order of evaluation for operators is left to right. if you write `f() + g()` in C#, the language guarantees that `f()` will be evaluated first. The example in the linked-to page is even clearer. The expression `F(i) + G(i++) * H(i)` is evaluated as if it were written like this:

```
temp1 = F(i);
temp2 = i++;
temp3 = G(temp2);
temp4 = H(i);
return temp1 + temp3 * temp4;
```

The side effects of each part of the expression take effect in left-to-right order. Even the order of evaluation of function arguments is strictly left-to-right.

Note that the compiler has permission to evaluate the operands in a different order if it can prove that the alternate order of evaluation has the same effect as the original one (in the absence of asynchronous exceptions).

Why does C# take a much more restrictive view of the order of evaluation? I don't know, but I can guess.[2]

My guess is that the language designers wanted to reduce the frequency of a category of subtle bugs (in this case, order-of-evaluation dependency). There are many other examples of this in the language design. Consider:

```
class A {
 void f()
 {
  int i = 1;
  if (true) {
   int i = 2; // error - redeclaration
  }
 }
 int x;
 void g()
 {
  x = 3; // error - using variable before declared
  int x = 2;
 }
}
```

The language designers specified that the scope of a local variable in C# extends to the *entire* block in which it is declared. As a first consequence of this, the second declaration of `i` in the function `f()` is illegal since its scope overlaps with the scope of the first declaration. This removes a class of bugs that can be traced to one local variable masking another with the same name.

In the function `g()` the assignment `x = 3;` is illegal because the `x` refers not to the member variable but to the local variable declared below it. Notice that the scope of the local variable begins with the entire block, and *not* with the point of declaration as it would have been in C++.

Nitpicker's Corner

[1]This is a simplified definition of *sequence point*. For more precise definitions, consult the relevant standards documents.

[2]I have not historically included the sentence "I don't know but I can guess" because this is a blog, not formal documentation. Everything is my opinion, recollection, or interpretation. But it seems that people take what I say to establish the official Microsoft position on things, so now I have to go back and add explicit disclaimers.

Raymond Chen

Follow