Why does the assignment operator in C# evaluate left to right instead of right to left?

devblogs.microsoft.com/oldnewthing/20170718-00

July 18, 2017

Raymond Chen

When I noted some time ago that the compound assignment operator is guaranteed to evaluate its left hand side before the right hand side, there was much commenting about why the language chose that model instead of the more "obvious" evaluation order of right-then-left.

In other words, instead of rewriting `E1 += E2` as `E1 = E1 + E2`, rewrite it as

```
temp = E2;
E1 = E1 + temp;
```

(Or as

```
((System.Func<T2, T1>)((e2) => E1 = E1 + e2))(E2)
```

if you want to keep it as a single statement.)

Thank goodness you can't overload the `+=` operator, because it would require that the operator overload for `+=` *be declared backward*:

```
operator+=(T1 y, T2 x)
{
 return x = x + y;
}
```

in order to ensure that the right hand side is evaluated first. (Because function parameters are evaluated left to right.)

(We'll come back to the rewrite rules later.)

One reason for the existing rule is that it keeps the rules simple. In C#, all expressions are evaluated left to right, and they are combined according to associativity. Changing the rules for assignment operators complicates the rules, and complicated rules create confusion. (See, for example, pretty much every article I've written about Win32 programming titled "Why does…?")

In particular, for compound assignment, it means that `E1 += E2` and `E1 = E1 + E2` are *no longer equivalent* if `E1` and `E2` have interacting side effects. Collapsing `x = x + y` into `x += y` would no longer be something you could do without having to think really hard first. Like hoisting closed-over variables, this would create another case where something that at first appears to be purely an issue of style turns into a correctness issue.

One argument for making a special rule is that any code which relied on `E1` being evaluated before `E2` is probably broken already and at best is working by sheer luck. After all, this is the rationale behind changing the variable lifetime rules for closures that involve the loop variable.

But it's not as cut-and-dried that anybody who relied on order of evaluation was "already broken".

Consider a byte code interpreter for a virtual machine. Let's say that the `Poke` opcode is followed by a 16-bit unsigned integer (the address to poke) and an 8-bit unsigned integer (the value to poke).

```
// switch on opcode
switch (NextUnsigned8())
{
...
case Opcode.PokeByte:
  memory[NextUnsigned16()] = NextUnsigned8();
  break;
...
}
```

The C# order of evaluation guarantees that the left hand side is evaluated before the right hand side. Therefore, the 16-bit unsigned integer is read first, and that value is used to determine which element of the `memory` array is being assigned. Then the 8-bit unsigned integer is read next, and that value is stored into the array element.

Therefore, this code is perfectly well defined and does what the author intended. Changing the order of evaluation for the assignment operator (and compound assignment operators) would break this code.

You can't say that this code is "already broken" because it's not. It does exactly what it intended, and it does it correctly, and what it gets is guaranteed by the language standard.

Okay, you could have come up with something similar for capturing the loop variable: Some code which captures the loop variable and wants to capture the shared variable. So maybe it's not fair showing code which relies on the feature correctly, because one could argue that any such code is contrived, or at least too subtle for its own good.

But as it happens, most people implicitly expect that everything is evaluated left to right. You can see many instances of this on StackOverflow. They don't actually verbalize this assumption, but it is implicit in their attempt to explain the situation.

The C# language tries to avoid undefined behavior, so given that it must define a particular order of evaluation, and given that everywhere else in the language, left-to-right evaluation is used, and given that naïve programmers expect left-to-right evaluation here too, it makes sense that the evaluation order here also be left-to-right. It may not be the best style, but it at least offers no surprises.

With the right-to-left rule, you get a different surprise:

```
x.value += Calculate();
x[index] += Calculate();
```

If `x` is `null`, or if `index` is out of bounds, the corresponding exception is not raised until after the `Calculate` has occurred. Some people may find this surprising.

Okay, so maybe can still salvage this by changing the rewrite rule so that `E1` is still evaluated before `E2`, but only to the extent where the value to be modified is identified (an lvalue, in C terminology). Then we evaluate `E2`, and only then do we combine it with the value of `E1`. In other words, the rewrite rule is that `E1 += E2` becomes

```
    System.CompoundAssignment.AddInPlace(ref T1 E1, T2 E2)
```

where

```
T1 AddInPlace<T1, T2>(ref T1 x, T2 y)
{
  return x = x + y;
}
```

This still preserves most of the left-to-right evaluation, but delays the fetch of the initial value until as late as possible. I can see some sense to this rule, but it does come at a relatively high cost to language complexity. It's going to be one of those things that "nobody really understands".

Bonus chatter: Java also follows that "always evaluate left to right" rule. Dunno if that makes you more or less angry. See, for example, **Example 15.26-2-2: Value of Left-Hand Side Of Compound Assignment Is Saved Before Evaluation Of Right-Hand Side**. However, for some reason, Java has a special exception for direct (non-compound) assignment to an array element. In the case of `x[index] = y`, the range check on the index occurs after the right-hand side is evaluated. Again, this may make you more or less angry. You decide.

I have a second bonus chatter, but writing it up got rather long, so I'll turn it into a separate post.

Raymond Chen

Follow