



Brace Yourself

Gerard J. Holzmann

YOU'VE PROBABLY HEARD

enough statements of the type, “There are only two important things to worry about in software engineering ...,” which is followed by a seemingly random choice of current topics. My favorite choice is “performance and correctness.” The first issue concerns the right choice of algorithm; the second concerns how you check that you’ve implemented that algorithm correctly. Your application fails if it’s either too slow or too buggy to be useful.

Dan Bricklin pointed out that software engineering courses do a great job of teaching the traditional topics but seem to neglect the difficulty of building secure, robust code that’s easily maintainable, testable, and extendable.¹ He said,

For years we emphasized execution speed, memory constraints, data organization, flashy graphics, and algorithms for accomplishing this all. Curricula need to also emphasize robustness, testing, maintainability, ease of replacement, security, and verifiability.¹

The reason for this situation might be that, because formal proof is generally beyond reach, most ef-

forts to verify the correctness of an application are reduced to some best-effort trial-and-error process. Lionel Briand, who heads the Software Verification and Validation Laboratory at the University of Luxembourg, put it more bluntly:

I [...] believe that proving correctness, at least in the large, is an illusion. The best we can do, like in most other engineering disciplines, is to minimize the risk with available resources.²

I couldn’t agree more, so the remainder of this column is about one simple thing you can do to reduce the residual risks that may be hiding in your code.

Reducing Risk

Consider these two reasonably simple expressions:

$$a + b * c,$$

$$a \wedge b \% c \ll d.$$

You almost certainly know how the first one will be evaluated. Multiplication takes precedence over addition, so the multiplication operator will be evaluated first. You can make this clear by adding parentheses:

$$a + (b * c).$$

Are you just as certain you know how the second expression will be evaluated? You can look up the relative precedence of the operators and try to figure out whether the result matches what you expected or what the writer of the code likely intended. Can you add the parentheses without looking up the precedence rules?

C defines 15 levels of precedence for approximately 49 unary and binary operators. Even if you’ve been programming for a while, you might not remember the relative precedence of some of the more rarely used operators, such as $\wedge$, $\sim$, or $|$. This means it’s wise to always make the intended evaluation order explicit in every expression using more than one operator.

A classic type of C expression that continues to make victims is

$$a == b \& c.$$

The bitwise operator $\&$ has lower precedence than the Boolean operator $==$, so the expression evaluates to

$$(a == b) \& c$$

and not, as you might have expected,

`a == (b & c).`

The confusion here is that the bitwise operator `&` does have higher precedence than the assignment operator `=`. So, if you use an assignment here, the expression will indeed evaluate to

`a = (b & c),`

with or without the parentheses.

Be More Associative

The previous examples cause confusion only if you omit the parentheses that can make the precedence rules of C irrelevant. There's another class of problems with C expressions, though, that doesn't have to do with precedence but with associativity. Consider these two expressions:

`a * b / c,`
`a / b / c.`

For the first expression, you'd be forgiven if you think the evaluation order doesn't really matter, so no parentheses are needed. Multiplication and division have equal precedence, and in plain math, the evaluation would produce the same result no matter which operation you performed first.

But you'd be making a mistake. You're not doing math; you're using a computer to approximate math, and that can be very different. If the multiplication occurs first, the result might overflow. You can avoid this by handling the division first, but in that case, you might run into a rounding problem and an unnecessary loss of precision.

There's no general rule here to tell you the right evaluation order in each situation. Of course, if you write the first expression without pa-

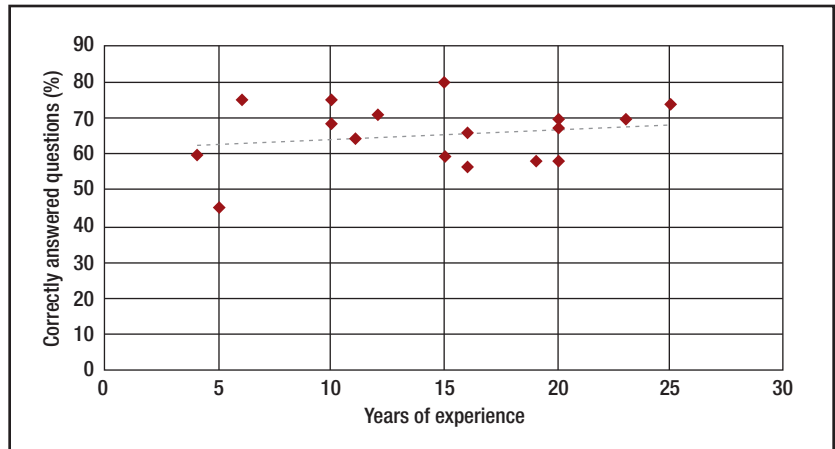


FIGURE 1. Results from Derek Jones's survey, in which participants had to insert parentheses in C expressions. The dashed line indicates the trend. If we omit the highest and lowest score, the remaining data points show no positive or negative influence of experience.

rentheses, what happens is well defined, although it could be surprising. Multiplication and division are left-associative, which means that without parentheses, the left-most operator is handled first.

You're not forgiven if you think the evaluation order of the second expression doesn't matter. Here again, no confusion exists about the precedence of operators; the associativity rules dictate that `100 / 10 / 10` will by default evaluate to 1, not 100.

The Jones Survey

A few years ago, Derek Jones did a survey of developers' intuitions of operator precedence and associativity rules in C.³ He asked 17 developers who attended the 2006 ACCU Conference (www.accu.org) to insert parentheses into computer-generated expressions to indicate how they would be evaluated. The survey involved 200 expressions, each with three different operands and two different operators. The challenge problems included expressions such as these:

`a % b ^ c,`
`a << b == c.`

Jones also asked each respondent how many years' experience he or she had in software development.

Although the sample size was very small, the tentative results were intriguing, to say the least. First, Jones measured how many problems the participants could solve in a fixed amount of time. The participants needed between 15 and 30 seconds per expression to insert parentheses, largely independent of their years of experience. This gives at least some indication of how much an expression without parentheses can slow you down when you're reviewing someone else's code.

Jones also plotted the number of correctly answered problems against the years of experience (see Figure 1). Not one participant answered all questions correctly. The highest score was 80.5 percent, the lowest score was 45.2 percent, and the average was 66.7 percent.

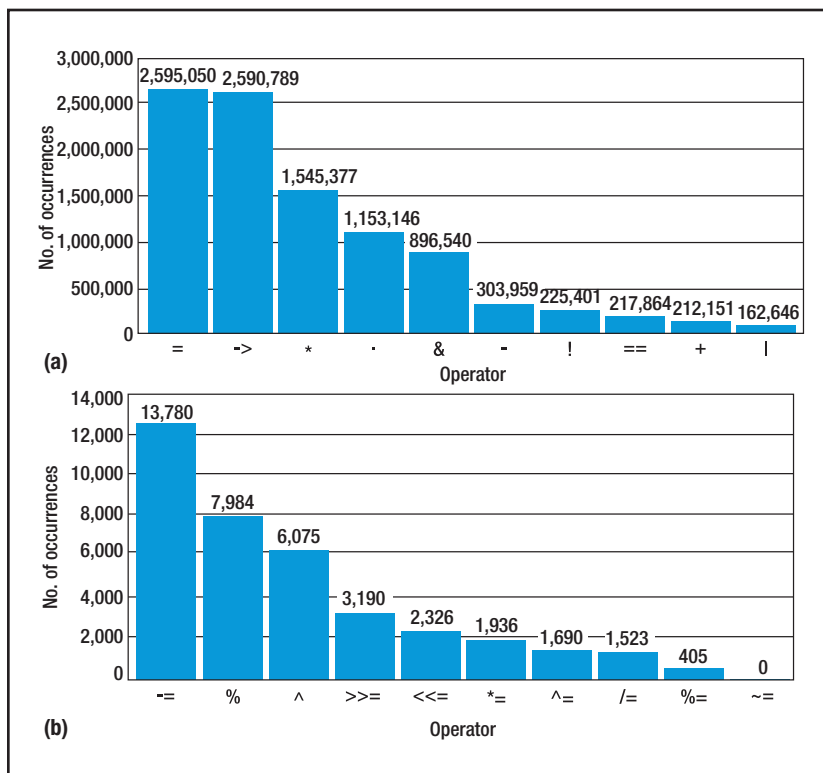


FIGURE 2. The (a) 10 most frequently used and (b) 10 least frequently used operators in Linux 4.3. Unfamiliarity with the less frequently used operators increases the risk of misinterpreting operator precedence levels and associativity rules.

This means that even the participant with the highest score estimated the relative levels of precedence and operator associativity incorrectly in about one of every five cases. On average, an astounding one of every three expressions was interpreted incorrectly. Although the trend line in Figure 1 suggests that the scores improve slightly as the years of experience increase, this is influenced heavily by the one “high” score of 80.5 percent. If we omit the highest and the lowest score, the remaining data points show no positive or negative influence of experience.

Assuming that this result is representative of what we would find in a larger study, it alerts us to our own frailty when it comes to remembering

obscure rules. Always using parentheses, even when it would seem obvious what the intent of an expression is, doesn’t necessarily make your code correct—but it can remove at least one likely source of incorrectness.

Frequency of Use

Most mistakes in Jones’ survey were for expressions with the bitwise operator `^`, followed at some distance by expressions with the modulo operator `%`. Unsurprisingly, perhaps, these two operators are among the least frequently used in most programs.

Figure 2 shows the 10 most and least frequently used operators in Linux version 4.3. The `^` operator shows up near the bottom of the list, with approximately 0.06 percent of

all uses. The `%` operator isn’t much more popular, at 0.07 percent.

I also checked how often `^` is used in combination with other bitwise operators in an expression, without parentheses indicating the desired evaluation order. A quick count showed just 128 such expressions—a very small number, given the amount of code. Just one of these expressions contains both `^` and `%`.

Those 128 expressions include some interesting cases. For example, how would you say

`a & b ? c ^ d : e`

will be evaluated? An expression like this appears on line 234 in file `./drivers/net/Ethernet/3com/3c509.c`. The binary operators `&` and `^` have higher precedence than the ternary operator `?`, so the resulting evaluation order is likely what the programmer intended. The use of combinations of `^`, `&`, and `|` can be tricky: `&` has higher precedence than `^`, but `|` has lower precedence. So,

`a & b ^ c`

means

`(a & b) ^ c,`

but

`a | b ^ c`

means

`a | (b ^ c).`

Undefinedness

There’s one final gotcha in writing C expressions that I want to mention. What value do you think the expression

$$(a = 1) + (a = 2)$$

will return? Assignment has lower precedence than addition, but the parentheses in this expression force the two assignments to be performed first. The order in which these assignments are executed determines the final result. That order, though, is officially undefined, which makes the whole expression undefined. So, technically, you can't even assume that the result will be 2, 3, or 4; it could be anything else as well.

Another example of an expression found in real-life code, which perhaps accidentally contained two side-effects on the same variable, is this beauty:

$$a = (b++ \ll 16) | b++;$$

In this case, one of the side-effects was hiding in a macro definition, so spotting it in the original code was

harder than in the version shown here. The compiler writer is free to return any value that's convenient when an undefined expression is used. So, don't assume that if one compiler produces a specific result, all compilers will match that result. You also can't assume that a future version of the same compiler will keep returning the same result or that the compiler will evaluate the expression the same way when a different optimization level is used.

Compilers typically don't warn about the presence of undefined or implementation-defined behavior, and neither do most current static source code analyzers. Grigore Rosu recently developed RV-Match, a tool for identifying cases of undefined behavior.⁴ It's based on a heroic effort to fully formalize the semantics of C as it's defined in the ISO C11 standard. RV-Match isn't foolproof yet; it's still in the experimental stage.

However, it could well become another important ally in our battle for more perfect, or should we say less risky, code. ☞

References

1. D. Bricklin, *Bricklin on Technology*, John Wiley & Sons, 2009.
2. "Interview with Lionel C. Briand," 2016; <https://issta2016.cispa.saarland/interview-with-lionel-c-briand>.
3. D. Jones, "Developer Beliefs about Binary Operator Precedence," 2006; www.researchgate.net/publication/242032246.
4. G. Rosu, "Defining the Undefinedness of C," *Proc. 36th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 15)*, 2015, pp. 336–345.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.

Take the CS Library wherever you go!



IEEE Computer Society magazines and Transactions are now available to subscribers in the portable ePub format.

Just download the articles from the IEEE Computer Society Digital Library, and you can read them on any device that supports ePub. For more information, including a list of compatible devices, visit

www.computer.org/epub