Editor: Gerard J. Holzmann
NASA/JPL
gholzmann@acm.org

Code Clarity

Gerard J. Holzmann

YOU PROBABLY WOULDN'T consider using a single letter for a global variable in a large application, nor would you think it wise to use an overly long name like `integer_loop_index_variable` in a simple `for` loop. Naming is important because it affects the readability of your code and the ease with which you can find your way around when reviewing code. Naming conventions aren't meant to help the compiler. A compiler has no trouble distinguishing names, no matter how long, short, or obscure they are. But to us humans, the names we use can matter a great deal.

> If this name growth trend always existed, we can try to calculate when the C epoch began.

Most coding standards contain naming conventions, and they often have rules that try to regulate the use of white space: where you put your comments, spaces, and braces. Clearly, if a C compiler doesn't care about the particular naming convention you follow, it cares even less about your use of white space. There's a difference between the two, though. Formatting choices are relatively easy to change after the fact with Unix tools such as `indent`. It's much harder to change a program to adhere to a different naming convention than the original programmer used, assuming of course that he or she used one in the first place.

If you write a lot of code, you're probably familiar with the problem that you have to quickly think of a name for a new function, while your main focus is to keep track of the logical flow of a new algorithm you're implementing. Who hasn't used quick names like `blurb()` or `doit()` and then forgotten to go back to replace those temporary names with something a little more meaningful? Like most programmers, I'm guilty of this. I did a quick check of the latest version of the Spin model-checker code (http://spinroot.com) that I maintain. Among the 643 function definitions, I saw function names such as `do_same()` and even the very helpful `blip()`. Clearly I can't be accused of having followed any particular naming convention there.

You often hear that the best function names are verbs and the best object names are nouns. The intuition is that from the function name you should be able to deduce what it does. The principle is fine as a starting point, but the problems begin if we try to make it a hard rule. For instance, would our programs improve if we renamed the familiar function `strlen()` as something like `compute_length_of_string()`? Similarly, would it be any clearer if we renamed `atoi()` as a verb-based variant like `convert_text_to_number()`? Besides knowing what a function does, it can be just as important to know what type of input a function takes and what type of result it returns. From a name like `strlen`, we can easily deduce that the function takes a string as an argument and returns an integer as a result. The fact that it must do a conversion to achieve that feat doesn't really need to be spelled out in this case.

Choosing Function Names

Some projects use the rule that all function names should start with an abbreviation of the datatype that the function returns and an indication of the function scope. We could, for instance, use the prefix `u32g_` for functions that return a 32-bit unsigned integer with global scope. In larger projects, we often also see the

0740-7459/16/$33.00 © 2016 IEEE

rule that function names must contain a prefix that includes the name of the module in which they're defined. This could extend the required prefix to something like `u32g_nav_`, and so on. For our innocent string length function, this could mean that its name could grow all the way to `u32g_gbl_compute_length_of_string()`. Proponents of this convention will likely say that it now makes it clear how to safely use the function and where its definition is. Meanwhile, we've already used more than 32 characters just to name one function.

Other than the name length, this approach has several problems. I'll name just three. First, if you're the programmer, you have to already know every part of the prefix before you can identify and use a function. If you don't, you have to look up the function definition first. Obviously, having the unknown module name be part of the unknown function-name prefix doesn't help you locate it faster than before. Second, if you have to look up the function definition or function prototype to determine the full name, it would hard to miss also seeing the function return type and scope. Third, there are surely better ways to check that your program is type-safe than using an easily violated naming convention. Are there extra penalty points for retaining the prefix `u32_` for a function that now returns a signed 64-bit result? Getting back to the uncontrolled growth of function name length: does this really improve code clarity?

When Longer Isn't Better

Should there then be a limit on function name length? The C99 standard requires that a C compiler distinguishes at least 31 significant initial characters in all names of functions and identifiers. The standard wisely also recommends that "implementations should avoid imposing fixed translation limits whenever possible."[1] Most, if not all, C compilers have taken this recommendation to heart and don't impose any limit on the number of significant characters in identifier names.

Technically, though, it can be considered implementation dependent what happens if you have two identifier names that differ only in the part that follows a common prefix of 31 or more characters. Many coding standards for safety-critical code therefore rule out identifier names longer than 31 characters. Thirty-one characters is enough to distinguish 26^{31} or 7.3×10^{43} different names, which is enough to give 10^{25} different names to every one of the estimated 7.5×10^{18} grains of sand on earth. So why would you ever need a longer name?

To get a point of reference for the types of naming conventions practitioners follow, I looked at the length of function names in the most recent Linux distribution (version 4.3). Figure 1 summarizes the results. I counted 390,312 distinct function names in the roughly 15 million LOC (18.6 million including the header files). Of those names, 97.5 percent are 31 characters or fewer. The remaining 2.5 percent cover 9,766 functions with names longer than 31 characters. The two longest names have 65 characters.

We can also look at earlier Linux versions to see whether the naming discipline has changed much over the years. To do this, I went back to Linux 2.4.18 from around 2002. That version had a mere 3.7 million LOC and 60,834 functions. As we know, code always grows with time, no matter what the application is.

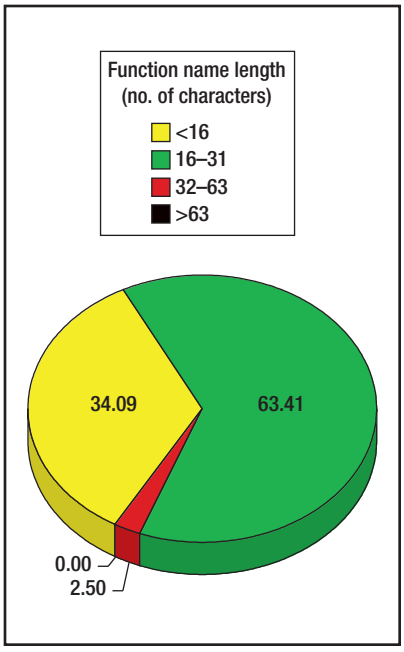

FIGURE 1. The length of function names in the Linux 4.3 source code. (The numbers in the chart are the percentages for each section.) A very small portion of the functions have names longer than 31 characters, with two functions reaching 65 characters. On the other end of the spectrum, 15 functions have a single-letter name. Most of these functions are single-line functions or dummies that just return a fixed value to the caller.

This earlier code had a different distribution of function name lengths. In it, 99.5 percent of the functions had a name shorter than 32 characters, and just 310 functions had names longer than 31 characters. The two longest were 50 characters. Those names are interesting though, because they differ only in the last four characters:

```
idetape_onstream_space_over_filemarks_
forward_fast(…)

idetape_onstream_space_over_filemarks_
forward_slow(…)
```

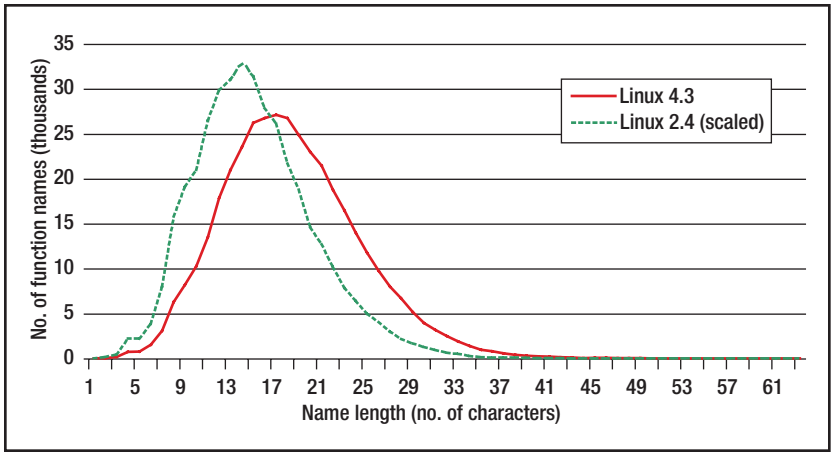

FIGURE 2. The distribution of function name lengths in two versions of Linux. I scaled the counts from Linux 2.4.18 to match the total from Linux 4.3 so that the two curves are more readily comparable. (Linux 2.4.18 defined 60,834 functions; Linux 4.3 has 390,312.)

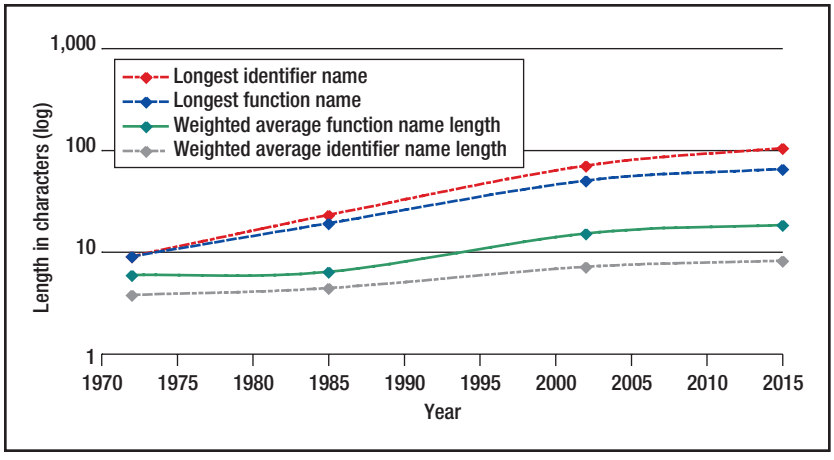

FIGURE 3. Finding the big-bang moment of the C epoch (log scale). The graph illustrates how name lengths have grown. The left-most data point is for the source code of the oldest preserved C compiler. The second data point is for the source code in the 8th Edition Unix **cmd** and **libc** subdirectories.

We can see the effect if we plot the distribution of function names used in these two Linux versions (see Figure 2). Over time, a shift occurred toward using not just more code but also longer names. In the 13 years from version 2.4 to 4.3, the median function name length increased from 14.0 to 17.5.

The growth trend is still more pronounced if we look at identifier names in general and not just function names. For instance, the longest identifier name in Linux 4.3 reached a whopping 104 characters.

Perhaps unsurprisingly, we see the same trend in the flight software NASA develops for its space missions. Over time, longer and longer identifier names have tended to get used, with some particularly striking extreme name lengths. For instance, the longest function name in the flight code for a recent mission reads almost like a sentence at 71 characters. Over approximately 18 years, the median function name length in flight code has increased from 18 characters in 1997 to 21 characters today.

So if this trend has always existed, we can try to calculate when the C epoch began, just like we can calculate the moment of the big bang by using the known rate of expansion of the universe to reason backward to the time when all mass would have had to originate from a single point.

The minimum function name length is obviously one character, but that would give us only 26 functions to play with. So, it's perhaps better to put the minimum length at two characters. If we put the average growth rate of function names optimistically at one character every three years and take the current median name length of 17.5 in Linux code, then time must have begun about $(17.5 - 2.0) \times 3 = 46.5$ years ago. That gives us a date in 1969.

Although this exercise stretches the limits of what's reasonable, the date actually turns out to be close to correct. The development that ultimately led Dennis Ritchie to the early design of C was Ken Thompson's B language, which itself was a derivative of Martin Richards's BCPL (Basic Combined Programming Language) from approximately 1969. The oldest version of a C compiler, which Dennis wrote in 1972 for the PDP-11/20 minicomputer, has survived, so we can look at that code to see how function names were

chosen then. The median length of the 52 function names in the 1972 C compiler was a little over six characters, with no function name being longer than nine characters or shorter than three.

Let's pick one more data point in 1985, with the source code from the `cmd` and `libc` subdirectories in the 8th edition of the research version of Unix. This code had 3,432 function definitions with, surprisingly perhaps, a median length of still no more than six characters. By this time, the longest function name had grown to 19 characters. There's a slight rub here, though, in that the long function names were actually contributed by colleagues from outside the Unix group. For the code written by people from the Unix group itself, the longest function name was still no more than 14 characters.

We can do a similar experiment by also measuring the average and longest lengths of all identifier names. The numbers are now much larger, of course. The 1972 C compiler code had 2,845 unique identifiers, the 1985 Unix code had 592,416, and the 2002 and 2015 Linux code had 3,821,169 and 20,961,560, respectively. The longest identifier in these code bases grew from 9 to 23 to 70 to 104 (see Figure 3). Figure 3 shows that, happily, unlike the rate of expansion of the universe, the rate of increase of name lengths isn't accelerating, so we may be reaching a new equilibrium.

How you name functions and identifiers clearly affects code clarity, but it would be hard to capture this fact into a single rule or a simple naming convention that you could apply uniformly to all code. It comes down to the judgment of the programmer whether a verb or a noun best captures the intent of a function. Clearly, very long names should be rare, and very short names are best for things that don't require much attention. Beyond that, there's little anyone could sensibly say about naming.

The statement "640 Kbytes of RAM should be sufficient for anybody" is often attributed to Bill Gates, spoken when PCs with 64 Kbytes of RAM were common. We could say that the implicit supposition from the C language standards that "31 characters ought to be sufficient for any identifier" has similarly been overtaken by reality.

Reference

1. "Translation Limits," *International Standard ISO/IEC 9899, Programming Languages—C*, American Nat'l Standards Inst., 2nd ed., May 2000.

GERARD J. HOLZMANN works at the Jet Propulsion Laboratory on developing stronger methods for software analysis, code review, and testing. Contact him at gholzmann@acm.org.