Code Clarity

Gerard J. Holzmann
Jet Propulsion Laboratory, California Institute of Technology

You probably would not consider naming a global variable with a single letter 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 in your code. Naming is important because it affects the readability of your code and the ease with which you can find your way around when you have to review that code. Naming conventions are not 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 it can matter a great deal.

Most coding standards have rules that dictate what naming conventions you should follow, and often they also have rules that attempt to regulate the use of white-space: where you put your comments, spaces, and braces. Clearly, if a C-compiler does not care about the naming convention you adopt, it cares even less about your use of white space. There is a difference between the two though. Formatting choices are relatively easy to change after the fact with Unix tools like **indent**. It is 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 are 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 flow of a new algorithm that you're implementing. Who hasn't used quick names like **blurb()** or **doit()** and then forgot to come back to it later to replace these temporary names with something a little more meaningful? Like most, I'm guilty of this. I did a quick check of the latest version of the Spin model checker code that I maintain. Among the 643 function definitions, I see function names like **do_same()** and even the very helpful **blip()**. Clearly I can't be blamed for following any particular naming convention there.

You often hear that the best function names are verbs and the best object names 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()** to something like **compute_length_of_string()**? Similarly, would it be any clearer if we renamed a function like **atoi()** to into a verb-based variant like **convert_text_to_number()**? Other than 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 has to do a conversion to achieve that feat does not really need to be spelled out in this case.

Choosing Function Names

How about the rule that all function names should start with an abbreviation of the type that the function returns, and an indication of the function scope. The prefix **u32g_** could be used for functions that return a 32-bit unsigned integer with global scope. In larger project we often also see the rule that function names must contain a prefix that includes the name of the module in which they are defined. This could then further 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 the function can safely be used, and where its definition can be found, but we've now already used more than 32 characters just to name a single function.

Other than the length of the name, there are a couple of problems with this approach. I'll name just a few. First, if you are the programmer you have to already know every part of the prefix before you can use a function. If you do not, you have to look up the function definition first. Obviously, having the unknown module name be part of the unknown function-name prefix does not help you to locate it and 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 seeing also the function return type and scope. Third, there are surely better ways to check that your program is type-safe than the use of an easily violated naming convention. Are there extra penalty points for retaining a prefix **u32_** for a function that now returns a signed 64 bit result? Getting back to the uncontrolled growth in the length of function names: can it really be argued that this improves code clarity?

When Longer is not Better

Should there then be a limit on the length of a function name? The C99 standard requires that a C compiler distinguishes at least 31 significant initial characters in all names of functions and identifier. The standard wisely also recommends that "implementations should avoid imposing fixed translation limits whenever possible." Most, if not all, existing C compilers have taken this recommendation to heart and do not 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 characters. Many coding standards for safety critical code therefore rule out identifier names longer than 31 characters altogether. 31 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?

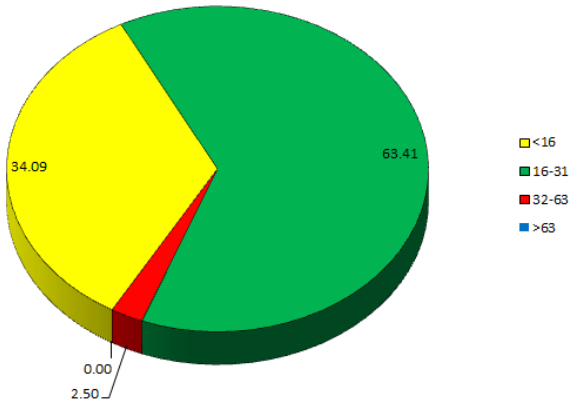

Figure 1 Length of function names in the Linux 4.3 source code. A very small portion of the functions have names longer than 31 characters, with two functions reaching a length of 65 characters. On the other end of the spectrum, there are 15 functions in the code that have a single-letter name. Most of these functions are single-line functions, or dummies that just return a fixed value of 0 to the caller.

To get a point of reference for the types of naming conventions that are followed in practice, we looked at the length of function names in the most recent Linux distribution (version 4.3). I count 390,312

distinct function names in the roughly 15 million lines of code (18.6 million if we include also all header-files). The large majority of those names, 97.5% to be precise, are 31 characters or shorter. The remaining 2.5% of the function names covers 9,766 functions that have names longer than 31 characters. The two longest count 65 characters.

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

We see a different distribution of function name lengths in this earlier code. This time 99.5% of the functions have a name shorter than 32 characters, and there are just 310 functions with names longer than 31 characters. The two longest are 50 characters. These names are interesting though, because they do differ only in the last four characters. They are:

```
idetape_onstream_space_over_filemarks_forward_fast(…)
idetape_onstream_space_over_filemarks_forward_slow(…)
```

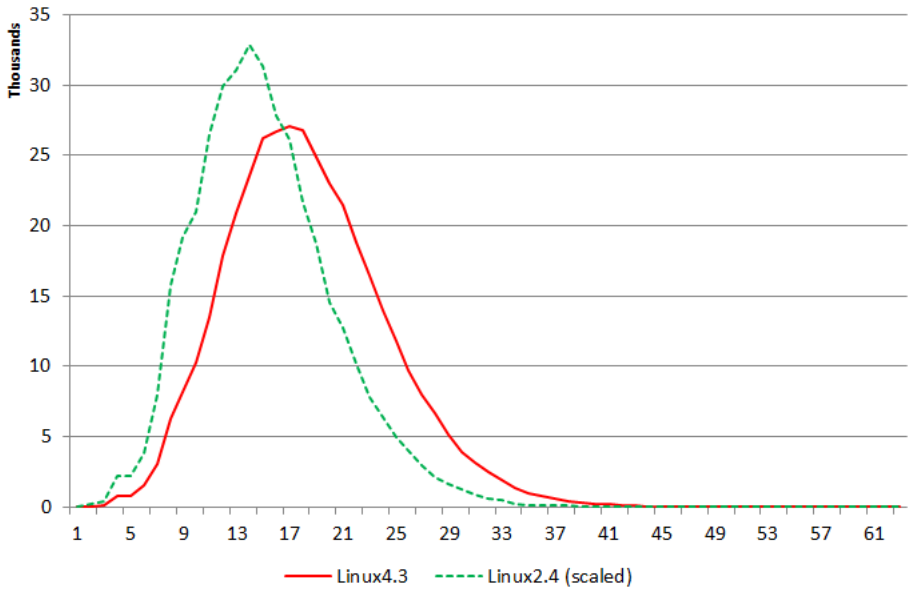

Figure 2 Distribution of all function name lengths in two versions of Linux. The counts from Linux 2.4.18 were scaled to match the total from Linux 4.3, so that the two curves are more readily comparable. (Linux 2.4.18 defines 60,834 functions, while Linux4.3 has 390,312.)

The effect can be seen clearly if we plot the distribution of function names used in these two releases of the Linux source code, as shown in Figure 2. Over time, there is a shift towards the use of not just more code but also longer names. In the 13 years from version 2.4 to 4.3 the median function name length in the Linux source code increased by from 14 to 17.5.

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

Perhaps not surprisingly, we see the same trend in the flight software that is developed at NASA for its space missions. Over time, longer and longer names tend to get used, with some particularly striking extreme name lengths. The longest function name in the flight code for a recent mission, for instance, reads almost like a sentence at 71 characters. Over a period of approximately 18 years, the median function name length in flight code increased from 18 characters in 1997 to 21 characters today.

If this trend always existed, we can try to calculate the beginning of time of the C-epoch…

So if this trend always existed, we can try to calculate the beginning of time of the C-epoch, just like we can calculate the moment of the Big Bang by using the known rate of expansion of the Universe to reason backwards to the point where 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 is perhaps better to put the minimum name length at two characters. If we put the average growth-rate of function names optimistically at one character for every three years, and we take the current median name length in Linux code of 17.5, then time must have begun about (17.5-2.0)*3 = 46.5 years ago. That gives us a date in 1969. Although this exercise stretches the limits of what is reasonable, the date actually turns out to be close to correct. The development that ultimately led Dennis Ritch to the early design of the C programming language was Ken Thompson's B language, which was itself a derivative of BCPL from approximately 1969. The oldest version of a C compiler, written by Dennis in 1972 for the PDP-11/20 has survived so we can look at that code to see how function names were chosen at that time. The median length of the 52 function names that appear in 1972 version of the C compiler is a little over six characters, with no function name being longer than nine characters or shorter than three.

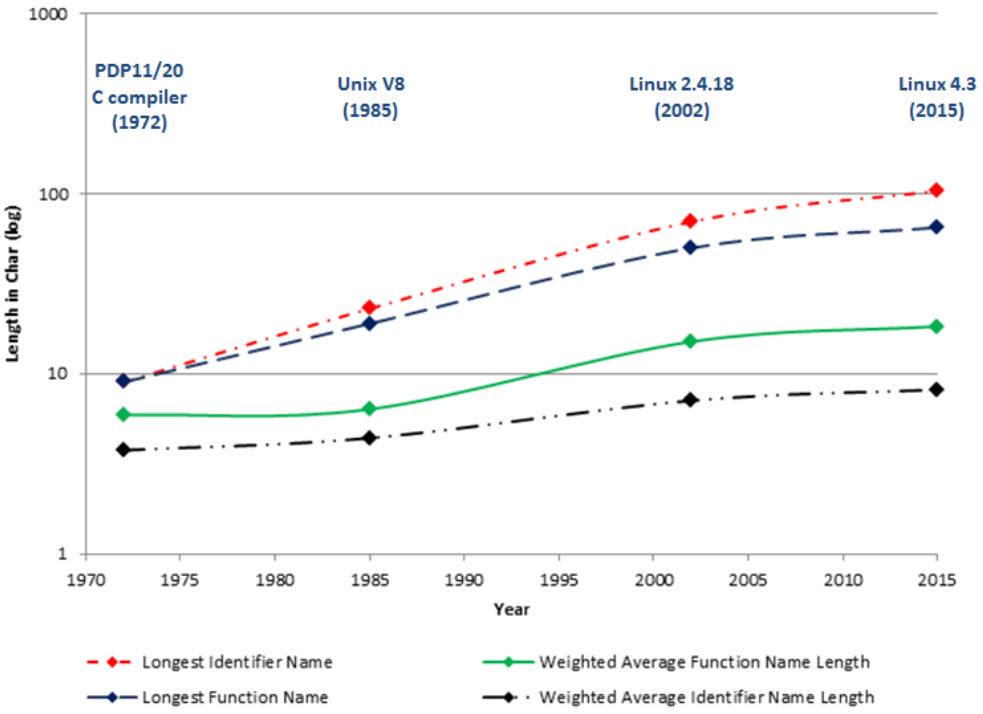

Figure 3 Finding the Big Bang Moment of the C-Epoch (log-scale).
This chart plots the weighted average and longest of all function names and identifier names found in the C sources of a series of code bases. The left-most data point is for the source code of the oldest preserved C compiler for the PDP-11/20 from 1972. The second data point is for the source code in the cmd and libc sub-directories of the 8th Edition version of Unix from 1985, and the next two data points are for Linux versions 2.4.18 and 4.3 code from 2002 and 2015 respectively. The graph illustrates how name lengths are growing. The C standard requires compilers to only distinguish names up to 31 characters long, although most existing compilers impose no such limit.

We will pick one more data-point in 1985, with the source code from the cmd and libc directories in the 8th Edition release of the research version of Unix. There are 3,432 function definitions in this code 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 is a slight rub here though in that the long function names were actually contributed by colleagues from outside the Unix group itself. For the code written by people from the Unix group proper the longest function name was still no more than 14 characters.

We can do a similar experiment to measure the average and longest lengths of all identifier names. The numbers are now much larger of course. In the C compiler code from 1972 there are 2,845 unique identifiers; in the Unix code from 1985 there are 592,416, and in the Linux code from 2002 and 2015 there are 3,821,169 and 20,961,560 respectively. The longest identifier in these code bases grows from 9 to 23 to 70 and then to 104, as illustrated in Figure 3. Figure 3 shows that unlike the rate of expansion of the Universe, happily, the rate of increase of function name lengths is not accelerating, so we may soon be reaching a state of equilibrium.

How you name functions and identifiers clearly has an impact on code clarity, but it would be hard to capture this simple fact into a single rule or a simple naming convention to could be applied uniformly to all code. It comes down to the judgment of the programmer if 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 used for things that do not require much attention. Beyond there is very little one could sensibly say about naming.

The statement “640KB of RAM should be sufficient for anybody” is often attributed to Bill Gates, spoken at a time when PC’s with 64KB 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 long since been overtaken by reality.

Acknowledgement

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.