How Not to Write FORTRAN in Any Language

DONN SEELEY, WIND RIVER SYSTEMS

There's no obfuscated Perl contest because it's pointless.
—Jeff Polk

Whatever language you write in, your task as a programmer is to do the best you can with the tools at hand. A good programmer can overcome a poor language or a clumsy operating system, but even a great programming environment will not rescue a bad programmer. —Kernighan and Pike

Programmers have debated the merits of different programming languages since the dawn of programming. Every coder has a favorite general-purpose programming language, and many have an unfavorite language, too. If the coder is old enough, often that unfavorite language is Fortran. The world has seen so much bad Fortran code that the name of the language is now a synonym for bad coding. Many of us have never seen real Fortran code, but we know what coders mean when they say, "You can write Fortran in any language."

I spent a significant part of my career in proximity to Fortran. Believe it or not, you can write good Fortran, as well as bad Fortran. No one would want to program in Fortran today, since many better alternatives are available. But you can write a usable and maintainable program in Fortran in spite of its many hindrances.

There are characteristics of good coding that transcend all general-purpose programming languages. You can implement good design and transparent style in almost any code, if you apply yourself to it. Just because a programming language allows you to write bad code doesn't mean that you have to do it. And a programming language that has been engineered to promote good style and design can still be used to write terrible code if the coder is sufficiently creative. You can drown in a bathtub with an inch of water in it, and you can easily write a completely unreadable and unmaintainable program in a language with no `gotos` or line numbers, with exception handling and generic types and garbage collection. Whether you're

There are characteristics of good coding that transcend all programming languages.

How Not to Write FORTRAN in Any Language

writing Fortran or Java, C++ or Smalltalk, you can (and should) choose to write good code instead of bad code.

LINGUISTIC DETERMINISM IS OVERRATED

Human beings do not live in the objective world alone, nor alone in the world of social activity as ordinarily understood, but are very much at the mercy of the particular language which has become the medium of expression for their society. —Edward Sapir

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. —Christopher Alexander

I'm going to state my biases up front, then attempt to justify them by example.

There is a famous view of linguistic determinism known as the Sapir-Whorf hypothesis, framed by Edward Sapir and his student Benjamin Lee Whorf. Roughly, it holds that the vocabulary and syntax of our language guide and limit the way we see the world: form dictates content. Edsger Dijkstra believed that programming in Fortran or Basic not only condemned us to produce bad code, it corrupted us for life.

The idea of programming-language determinism has some truth to it, but is overrated. Because we are often tackling the same problems in C, Perl, Scheme, Smalltalk, and so on, we can usually find a way to analyze them and code solutions using common designs. Sometimes the features of a particular language make a particular solution much more elegant and comprehensible, and in that case form influences content. But these languages have enough common ground that they can share many designs. C may have pre-increment and post-increment operators, but you can still add 1 to a variable in any language that supports variables.

C doesn't contain built-in support for features such as object-oriented programming. You can still use an object-oriented *design* in C, however, if the design is fairly simple and solves your problem. Many data structures and libraries written in C have the form and function of objects, even though the language does not directly support inheritance, packages, private members, and other features. The `vnode` structure is a fine example. The idea of the `vnode` abstraction layer for file system operations became so popular among Unix-like operating systems because it is such an obviously good design and because C didn't get in the way, although it didn't provide any help either. My experience with other ideas and other languages has been similar—if the design is good, and the language doesn't get in the way, then programmers will adopt it and adapt it. Many good programming ideas can be used a million times over in different languages and operating system environments.

WHY WE WRITE CODE

You are not expected to understand this. — Comment above a nonlocal `goto` in `swtch()` in Version 6 Unix

We can make some generalizations about good code across different general-purpose programming languages because there are common reasons why we write code,

```
case 5:
#line 60 "/usr/src/core_contrib/linux/transform/parse.y"
{ puts(yyvsp[0].string); }
break;
case 14:
#line 72 "/usr/src/core_contrib/linux/transform/parse.y"
{ add_alias(&yyvsp[-3].ident, &yyvsp[-1].ident); }
break;
case 15:
#line 76 "/usr/src/core_contrib/linux/transform/parse.y"
{ push_file(yyvsp[0].string); }
break;
```

FIG 1

including the following two big reasons.

First, we write code to design solutions to problems. Therefore, good code has good design elements. In the real world, many problems that are similar to yours have already been solved, so there usually are plenty of good design elements that you can use to compose into your solution. The success of the solution follows from the quality of the design; bad design often leads to wrong "solutions" (aka bugs). The design may not scale well, or it may be difficult to extend, or it may be so complex that finding and repairing bugs becomes an extremely painful and time-consuming task. Good design is easy to identify because it seems intuitive and (often) familiar.

Second, we write code to communicate our designs to people. Machines are interested only in ones and zeroes. They couldn't care less about a high-level programming language. People, on the other hand, do quite poorly when maintaining code written directly as ones and zeroes. For people, form and style are very important to communicating content. Good form is transparent: the reader "sees" the content, rather than the form or style. Good form is also elegant. It doesn't clutter the page or confuse your mind with excess information. Your reader will most thoroughly assimilate your code when it is both easy to understand and contains few distractions.

If you are skeptical that you write code for people rather than for machines, I suggest that you take a look at machine-generated code: (See Figure 1)

If you wrote and maintained code like this, you would go nuts very quickly. What do all those numbers mean? The machine knows, but we don't. Form and style are extremely important to good code, much more important than many programmers believe they are.

HOW NOT TO WRITE BAD CODE

The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. —Edsger Dijkstra

My ideal is that I should be able to read through good code and find that the design is so obvious and intuitive, and the style so natural and transparent, that I can characterize it accurately in my native language more or less in realtime. I've spent almost 30 years maintaining code, most of which was written by someone else, and by now I know exactly what I want to see, and I try to put that knowledge into my own code.

Elements of good design have been described well elsewhere, so I'm going to emphasize form and style here, particularly those features that apply across programming languages. Many of the features that I look for in good code can be found in every general-purpose programming language, even Fortran. Here are some of them.

GROUPING

Every modern general-purpose programming language provides great leeway over white space. It is typically mandatory only between identifiers, and optional around punctuation and operators. Blank lines, indentation, and spacing within expressions are all permitted but are all optional. In spite of this, white space is mandatory in good code because it implements grouping.

When we write natural languages, we use white space as a form of punctuation, to mark paragraph breaks and headings. White space has the same function in coding. Of course, the compiler couldn't care less:

```
This

                                    =

  is

 +

                  very

  *

 annoying

;
```

You should use blank lines and indentation in code to emphasize its logical structure, as a guide to the reader. If the code is nicely paragraphed using blank lines, it's far easier to navigate. Insufficient white space makes it hard to see structure and, almost as importantly, makes it hard to follow code on the screen or on the page. If you briefly look away from a solid chunk of code, say, to check on a variable declaration, it can be tough to find your place when you look back. If the programming language lets you omit spaces around operators and punctuation, you can get the same effect within a single line of code:

```
This=is+almost*(as-annoying);
```

You can use internal white space on a line to emphasize logical sections, such as the left and right sides of an assignment:

```
This = is + much * (more - readable);
```

Abuse of white space creates visual clutter, which, ironically, causes the same problem as lack of white space: the

How Not to Write FORTRAN in Any Language

text becomes hard to navigate because the structure and the landmarks disappear. Most of the bad code that I've seen, however, seems to err on the side of too few blank lines. Over the course of time, I have found my own code having more blank lines for grouping. I try not to let it get too choppy. It's a bit like the difference between paragraphs in a newspaper and paragraphs in a novel—form needs to be transparent in a newspaper, whereas a novel is often a deep relationship between form and content.

Let's go a little deeper. Machines don't care how long a group is. So why should we? People have a limited amount of short-term memory. It holds maybe seven items. (This is supposedly why local phone numbers in the U.S. have seven digits.) You can get around this limit through *chunking*. If you can group several items together, the group itself becomes an item. People can build hierarchies of groups in their minds, but at each level, they will remember things better if there are a limited number of items. If your code exceeds the limit, people get confused and start fumbling.

This mental limitation hits at every level in code, all the way down to statements. If a line of code is jammed with operations, it becomes difficult to read and understand. I have seen plenty of code that uses really wide lines or shorter indentation stops to pack more information on each line. I think that this is wrong. More code per line is not a virtue; readability is a virtue.

Statements that creep from several elements to many elements need to be broken into multiple statements. Most modern languages permit line breaks in the middle of statements, but the line breaks don't make the statements easier to read. Sometimes an interface dictates the number of elements in a statement, and you're pretty much stuck. Even then, you could minimize the pain:

```
Can(you, tell, at + a, glance, which * of, these,
parameters, is(the, eighth), one ? yeah : sure);
```

When you have this many parameters, it's probably time to switch to an interface that passes a record rather than individual parameters.

Familiarity and patterns reduce the memory load for an item in a group. A visual pattern can make code strikingly simple to read:

```
It = 0;
is = 0;
pretty = 0;
obvious = 0;
what = 0;
this = 0;
code = 0;
does = 0;
even = 0;
at_this = 0;
length = 0;
```

You can take advantage of this memory trick by keeping the groups as natural as possible. You can put blank lines around the following:

- Related field declarations in class or record declarations
- Related constants
- Related local and global declarations
- Related variable and field initializations in code
- Related arithmetic statements
- Related output statements
- Related defensive programming elements

It's a simple principle: related things are easier to remember as a group.

Indentation also provides grouping. It's a visual indication that's different from blank lines, but related. I'm sure that all of us have seen bugs where indentation gave a very misleading sense of structure, even though indentation has no syntactic significance in the programming language. Good indentation practices are especially important in languages such as Lisp, where the visual cues for structure are more difficult to spot. Fortunately, the standard coding styles for various languages put a lot of emphasis on indentation, so there are fewer examples of indentation abuse these days.

Related to indentation is tabbing. I'm referring to the practice of aligning elements within a line with elements vertically above and below it. Tabbing appears to be a common practice, primarily serving to emphasize an important operator such as assignment. Wide tabbing, however, tends to make me visually associate elements in a vertical dimension rather than a horizontal one, even if the horizontal grouping is more natural:

```
I =             because;
scan =          these + items;
down =          are(associated);
the =           vertically;
column =        "so it feels";
rather =        natural;
than =          to(read, them);
across =        that % way;
```

This sort of code can be impressively hard to read correctly. Tabbing works best when the average horizontal separation is small. I usually avoid tabbing in code because of this problem, although tabbing is sometimes mandated by a coding standard.

Tabbing for comments is less of a problem. When one column is code and the other column is comments, they do have a natural grouping. It's a visual clue that the comments aren't code. When tabbed comments aren't close together, they stand out and can serve as visual landmarks. Even with comments, wide tabbing still tends to make me read down the columns rather than across. It also creates more visual clutter, concealing the structure of the program.

Natural grouping is so important for readability that it may be worth changing the structure of some code to group-related elements. Sometimes, deeply embedded control structures can split groups and even cause issues for the limit on memory. If you can flatten out the control structures by using alternative syntax or moving code into subroutines, the code can become much more readable.

This problem is evident in natural language. In linguistics, there is a well-known phenomenon called self-embedding that demonstrates the issue:

The WMDs that the UN inspectors that Iraq charged were spies failed to find were a figment of Bush's mind.

This is grammatical English and thoroughly unreadable. The "WMDs" "were a figment," but those phrases are visually far apart and are quite hard to match up. In code we can sometimes use indentation to make the grouping work:

```
if (WMDs != FIGMENT)
        if (WMDs == 0)
                if (spies(&inspectors) == TRUE)
                        dump(&Tenet);
                else
                        withdraw_from(UN);
        else
                invade(&Iraq);
 else
        seek(PSYCHIATRIC_HELP);
```

But this still breaks up natural groupings by putting the else clauses far from the if tests. Watch what happens to readability if we flatten out this code:

```
if (WMDs == FIGMENT)
        seek(PSYCHIATRIC_HELP);
 else if (WMDs != 0)
        invade(&Iraq);
 else if (spies(&inspectors) == FALSE)
        withdraw_from(UN);
 else
        dump(&Tenet);
```

Of course, you can also avoid excessive embedding by pushing the deeply nested code out to a subroutine.

COMMENTS

Comments are syntactically white space in modern general-purpose programming languages. The compiler ignores them, but the reader won't!

Everyone has a favorite bad comment.

```
/* add one to I */
i = i + 2;
```

Because no mechanical check exists for the correctness or appropriateness of comments, they are ripe for abuse. A comment that restates the obvious is just visual clutter. A comment that describes code incorrectly is a disaster for the reader. Comments of the first kind often turn into the second kind when someone makes a fix to the code and not to the comment.

A good comment shouldn't restate code; good code should speak for itself. Rather, a good comment should motivate or explain the code without introducing details that are properly part of the code. A bad comment:

```
/* shift x by 2 and add to base */
 result = base + (x << SCALE_FACTOR);
```

A better comment:

```
/Memory is partitioned.  Scale the index.  */
result = base + (x << SCALE_FACTOR);
```

I'm a strong believer that single- and multi-line comments should contain real sentences, rather than telegraphese or pseudocode. This may seem like a lot of work, but I also believe that once you are writing good code and avoiding bad comments, you'll find that fewer good comments will more than make up for lots of bad comments. People will appreciate real sentences in comments, since they won't have to struggle with yet another language. Reading code is hard enough already; why make readers do yet more work to understand comments?

Good code should use the "principle of least astonishment." Readers often miss a tricky spot if they haven't been warned to expect one. A nice block comment is a

How Not to Write FORTRAN in Any Language

fine way to flag code that needs closer inspection. Good code should avoid relying on tricks as much as possible, but when a trick is unavoidable, put up those orange cones and flashing lights:

```
 /*
 * If the new process paused because it was
 * swapped out, set the stack level to the last call
 * to savu(u_ssav).  This means that the return
 * which is executed immediately after the call to aretu
 * actually returns from the last routine which did
 * the savu.
 *
 * You are not expected to understand this.
 */
if (rp->p_flag&SSWAP) {
        rp->p_flag =& ~SSWAP;
        aretu(u.u_ssav);
 }
```

(No, that '=&' operator is not a typo.)

Comments can be important visual cues to structure. Lining up stars in C comments or semicolons in Lisp comments or octothorpes (#) in shell comments draws the eye. A visual theme like this also is a great way to make comments look distinct from code.

NAMES

What's in a name? Plenty. Good names are extremely important to good code. You get to pick most of the names that you use in code. Like comments, names mean nothing to the machine. They have no significance except as strings of characters that stand for elements of the program, such as variables, functions, classes, or more exotic things. Like comments, names have much more meaning to people. To make code readable, you need to choose names wisely.

What is important in a name? As we saw previously, familiarity reduces the mental workload, so familiar names in familiar contexts are easier to understand. Coders have used `i` as the name of an index variable since time immemorial. It's boring, but you can't go wrong using `i` for an index variable in your own code.

The more familiar the name, the more it communicates to the reader:

```
Int
main(int argv, char **argc)
{
[...]
}
```

If you know C, the preceding code will make you choke. Long ago, I actually was exposed to some C code that was deliberately written this way. In C, the names `argc` and `argv` are *not* arbitrary. If you see them in a function, even a function other than `main()`, you know exactly what they are supposed to mean. These names are not mandated by the ANSI C standard, but they are still standard practice. Ignore their standard usage at your (great) peril.

We need to consider this issue even for names that are not part of standard practice. If you use the name `pShl` for a local variable that points at an `SHL_NODE` structure, you should be consistent and never use that name for a different purpose anywhere else in the program. Even better, you should use the same name for the same purpose in the same context throughout your code. If you're consistent, then when people see `pShl` anywhere in your code, even without having seen the declaration for it, they will know exactly what it does. Reducing the burden on the reader's memory by using familiar names will make code immensely more readable.

I can't stress this enough. Wise name choice is maybe the biggest factor in writing readable code. Familiar and obvious names make code more readable. Use familiar and obvious names whenever possible and be consistent about the names you use.

Also, like comments, there is a tension in naming between descriptiveness and visual clutter. There is a balance between these extremes that I have managed to reach after reading and writing a lot of code. I try to use short, punchy names for commonly used elements and longer, more descriptive names for rarer elements:

```
char c;
off_t o2;
extern iso_t neodymium_148;
extern iso_t *actinide_series[14];
```

Really long names can get in the way so much that they obscure the structure of code. On the other hand, squeezing really long names down into acronyms can produce code that looks like line noise.

In spite of being lexical atoms, names are composable:

```
mol_t calcium_carbonate;
mol_t calcium_magnesium_carbonate;
mol_t potassium_magnesium_iron_aluminum_silicate_
hydroxide_fluoride;
```

This is another fine way to create familiarity. We want to use similar names for similar elements. In natural languages, we build words out of elements called *morphemes*, like *carbon* + *ate*. We do the same thing with names in code: off_t is off + _t. Composition can sometimes create awkward names:

```
void XrmStringToBindingQuarkList(const char *,
    XrmBindingList, XrmQuarkList);
```

As long as we're talking about exceedingly long names, I might as well mention my biases:

```
I_find_underscores_easier_to_read_than(lotsOfStudlyCaps);
```

Underscores are a bit like white space within identifiers. But reasonable people can differ on this weighty subject.

CONSISTENCY

Style guides: I hate 'em. After all, I know which style is the best: mine! Style guides often appear to be dreary lists of arbitrary-seeming rules that limit my creativity. Reading them puts me to sleep.

When I maintain code, however, I set aside my personal style and try to match the style of the project. I want my code to look exactly like everyone else's code, at least as far as the style guide goes. The reason for this, again, is familiarity. (Is this sounding familiar?) If you use the same coding conventions throughout a software project, the maintainers will grow accustomed to the style and it will magically become transparent to them. They will see the code, not the style.

Lack of consistency is one of the hallmarks of bad code. If 30 different people worked on a source file, I really, really don't want to see 30 different coding styles or naming schemes when I read it. It becomes a nightmare to attempt to find structure in code like that. Coders have to be humble and accept that for code to be readable, their favorite style is not as good as the established style.

To summarize, regardless of the programming language, good code should:

- Avoid clutter
- Use chunking
- Use familiarity
- Prevent astonishment
- Be consistent

I KNOW BAD CODE WHEN I SEE IT

It is practically impossible to teach good programming style to students that have had prior exposure to Basic; as potential programmers they are mentally mutilated beyond hope of regeneration. —Edsger Dijkstra

I'm not Dijkstra or Kernighan, but I've been coding since I was knee high to a wumpus. It's possible that my mind was destroyed by all the Basic that I wrote in high school. I can still remember discovering GOSUB and finding ways to use it in my (few) programs. I wrote my share of Fortran, too. I can also remember trying to force some of my Fortran practices on Algol when I encountered Algol for the first time. I ended up doing coding projects in many programming languages. After a while I developed an eye for what was common among them.

Like pornography, I know bad code when I see it. I usually know good code when I see it, too. I think most other coders do as well. And from reading lots of bad code (and some good code), I have come to realize that the code's programming language is less important to the quality of the code than the way in which the code is (ab)used. I think I have found a number of reasonable explanations for why some code looks good and some looks bad.

I still see a lot of bad code. There are plenty of excuses:

- The code was written under tight deadlines.
- It was someone's first big coding project.
- It was only supposed to be a prototype.
- It began as a personal project.

The effort required to write good code rather than bad code is really pretty small. The payoff for good code over time as various people maintain the software is really quite large; it just doesn't make sense to write anything other than good code right from the start. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

Donn M. Seeley is a senior member of technical staff at Wind River Systems. He was a co-founder of Berkeley Software Design, Inc., the first commercial vendor of 4BSD. He is the author of "A Tour of the Worm," one of the original papers on the Morris Internet Worm incident of 1988. He is currently working on embedded systems technology at Wind River Systems.

© 2004 ACM 1542-7730/04/1200 $5.00