Everything Functional Fortran Style Guide

August 26, 2020

1 Motivation

One key aspect of software development is that code needs to be readable by human beings, not just machines. The following quote emphasizes just how important human readability is.

> "Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write."
>
> – Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

In fact, code that cannot easily be read and understood quickly becomes hard to use and hard to change, making it useless code. Even if the code "works" today, what it does now isn't what it will always need to do, and if you can't change it, the code becomes obsolete.

While a good style guide may not be sufficient for ensuring your code is readable and understandable, it is almost certainly necessary. A good style guide, consistently followed, can help reduce the mental overhead required to read and understand code. When reading a section of code, you can find things where you expect them to be found, horizontal and vertical white space become visual cues that guide the eyes and reveal structure in the code, and naming conventions and capitalization style can embed even more detail without requiring additional verbosity.

Even when these little bits of mental overhead may seem small and inconsequential, I can promise you that they add up. When your trying to reason about and fit all of the complexities of some complicated code into your head, every little bit helps. So anything we can do to make this job even a little bit easier is time well spent.

Also included are some guidelines for software design and construction. In general they conform to industry standard practices and conventions. They are not intended to be dogmatic, but to focus on producing code that is readable, understandable and maintainable with a consistent style. There may be some instances where adherence to these guidelines would not achieve these goals, and in those instances you should do what produces the best code.

Figure 1: Visual Layout Example

> **NOTE:** What is readable and understandable should always be judged by someone other than the author. If you wrote the code, of course you'll be able to understand it. But it is far more important that others also be able to understand it. That other person may even be you six months, or even 10 years from now.

2 Visual Layout

The visual structure of code alone can communicate a wealth of information to the reader, without even focusing on the letters or words used. As an example, in Figure 1 where I've simply used rectangles in place of the text of the code, it immediately becomes recognizable as a procedure, with an argument list, declarations, and the body of the code with what is likely an if/else block. The layout communicates all this information without a single, legible character needed.

2.1 Indentation

Indentation is one of the most powerful tools available for using layout to convey the structure of code. I use 4 spaces for each level of indentation. This makes identifying different levels of indentation in multiply nested, or lengthy sections of code much easier. Each statement inside a block is indented one level more

Listing 1: Indentation Example

```
module something_m
    ...
contains
    ...
    function do_something(arg1)
        ...
        ...
        thing = &
                this_statement &
                is_really &
                long
        if (thing) then
            ...
        else
            ...
        end if
    end function
    ...
end module
```

than the statements that begin and end that block. This includes the beginning and ending statements of a program or module. This discourages the use of deeply nested code structure or overly long lines, which can be a sign of overly complicated code that should be refactored and simplified. An example of this structure is shown in Listing 1.

Continued lines are indented **two** levels more than the first line of a statement. This ensures that the statement is readily identified as being split across multiple lines, and there is visual distinction between the continued lines and subsequent statements, even if the following statement is indented (i. e. a long `if` statement is split across multiple lines as demonstrated in Listing 2).

Listing 2: Continuation Indentation Example

```
if ( &
        this == that &
        .and. other < another) then
    thing = doodad
    ...
```

2.2 Blank Lines

Blank lines are a great way to delineate sections of code without adding visual clutter. I use one blank line between procedures, and one blank line between the use statements and the declarations, and between the declarations and the executable statements. Strive to minimize any blank lines between executable

Listing 3: Blank Lines Example

```
module something_m
    use some_m, only: ...
    use another_m, only: ...

    implicit none
    private

    integer, parameter :: THE_ANSWER = 42

    public :: do_something
contains
    function do_something(arg1, arg2) result(answer)
        real, intent(in) :: arg1
        real, intent(in) :: arg2
        real :: answer

        answer = THE_ANSWER + arg1 - another_thing(arg2)
    end function

    function another_thing(arg3) result(another)
        real, intent(in) :: arg3
        real :: another

        another = arg3 * 2.0 / 3.0
    end function
end module
```

statements, as sections of logic inside a procedure call for breaking those sections out into their own procedures. Listing 3 illustrates the use of blank lines in some example code.

2.3 Line Length

It is widely agreed upon throughout the software community that there should be **some** limit on the length of lines. What that limit should be is highly debated, and usually boils down to a matter of personal preference and arguments about screen size. Historical values are usually based on limitations of early computer equipment, with screens only being able to display 80 characters, punch cards being limited to 72 characters, etc. However, we can look to research in the field of typography and ease of reading for some guidance on the matter.

For printed text it is widely accepted that line length fall between 45–75 characters per line (cpl), though the ideal is 66 cpl (including letters and spaces) [1]. For electronic text, studies have shown that optimal line length is a bit more complicated. For example, longer lines would then be better suited for cases

when the information will likely be scanned, while shorter lines would be appropriate when the information is meant to be read thoroughly [5]. As source code is generally meant to be read thoroughly, it would suggest we should try and keep line lengths to a minimum. More complicated lines also tend to be longer, and so by keeping line lengths shorter, we can keep complexity to a minimum.

I recommend setting your editor to identify column 80, and treating that as a strong suggestion. Lines that go past 80 by a few characters may be okay, but much more than that should be continued, or ideally split into separate statements and simplified.

2.4 Space Within a Line

White space used appropriately within a line can also make individual statements easier to read and understand. The spacing on a line should be as follows:

- There should be no space between the name of a procedure and the opening parenthesis
- There should be no space between an opening bracket/parenthesis and the first item inside it, or between the last item and the closing bracket/-parenthesis
- There should be no space before a comma or semicolon, and one space after
- There should be a space on either side of a binary operator, including the assignment operator (`=`) and comparison operator (i. e. `>=`). (i. e. `z = x + y`)
  - One exception would be in cases where multiple operators appear on the same line and the order of operations is important (i. e. `2*x + 3*y`). Be careful that the spacing is actually correct because something like `2 * x+y * 3` gives the impression to the reader that it means `2 * (x+y) * 3`, while the compiler still performs `2*x + y*3`
- There should not be a space between a unary operator and its operand (i. e. `x = -y`)

2.5 Breaking Long Lines

While it is recommended to keep line length and complexity to a minimum, sometimes long statements are inevitable. In those cases, a good convention can help ensure these statements are still easily understood. For statements that do not fit on one line, use the following guidelines for breaking/wrapping the statement onto multiple lines. The goal here is to break lines and align continued lines in such a way that related items appear together (i. e. the argument list for a function), lists appear in a single row (i. e. all on one line, or one item per line), and renaming of any item in the statement does not disrupt the alignment.

- When the name of a procedure and its arguments do not fit on the same line, break the line after the opening parenthesis and indent the following line(s) two levels. Either put all the arguments on one line, or one argument per line. See the examples in Listing 4.
- Break lines before operators (except the assignment operator) and indent the following line(s) two levels. See the examples in Listing 5.

Listing 4: Procedure Declaration Continuation Examples

```
function this_is_a_long_function_name( &
        with, lots, of, arguments) result(thing)
    ...

function this_is_another_really_long_name( &
        with_really_long, &
        argument_names, &
        on_multiple_lines) &
        result(thing)
    ...
```

Listing 5: Assignment Statement Continuation Examples

```
this_really_long_variable = &
        another_long_variable * something_else &
        + another_thing

this_variable = comes_from_a_function( &
        with, lots, of, arguments)

this_other_really_long_variable = &
        comes_from_another_function( &
                with_some_long, &
                argument_names)

this_super_duper_long_variable_name = &
        comes_from(short, things)
```

2.6 Procedure Length

Keep procedure length to a minimum. How long is too long? Pretty short. What's the ideal length? Shorter than that. I tend to try and stick to no more than 10 executable statements. If it is longer than that it is probably doing too much, or getting bogged down in the details. Extract sections of such code into their own procedures. This allows the code to read at a higher level of abstraction and reduces the complexity.

3 Names

Names are incredibly important. They are the best - if not only - way of communicating intent to readers. Take the time to pick good, descriptive names for things. Don't use uncommon or difficult to read abbreviations. Don't use the same words to describe things which are semantically different, and be consistent about using the same word to describe things which are semantically the same. My favorite guidance for selecting names comes from Chapter 2 of Clean Code [6].

Fortran has a few interesting characteristics which drive towards a more "noisy" naming convention. First, Fortran is case insensitive. Meaning `some_thing`, `Some_Thing`, `SoMe_ThInG`, and `SOME_THING` would all refer to the same thing. So while you can use capitalization to signify certain things to the reader, they are generally avoided because compilers and other tools generally convert everything to upper or lower case anyways, so any error messages may not necessarily match your code.

Secondly, all names live in the same "space". Meaning modules, programs, procedures, types and variables must all have different names. Scope still exists, so one module may have a type with some name, and a different module may have a variable with the same name, but within the same scope, a type may not have the same name as a variable.

Lastly, keywords in Fortran are not reserved words. Meaning it is possible (but not recommended) to have a variable named `if`. So something like Listing 6 would be horrible, but valid code.

Listing 6: Don't Do This

```
if (if) then
    then = else
else
    else = end
end if
```

3.1 Naming Convention

All names should be lowercase, with words separated with underscores. However, variables declared as `parameter` should be all caps. This is the one case where location, context and syntax don't quite sufficiently disambiguate, and the added clarity can be quite significant. Module names should be suffixed with `_m`, as it is quite frequent to have a procedure with the same name, or a type with the same name as the module. Type names should be suffixed with `_t`, as it is quite frequent to have a variable with the same name as the type.

There are three main roles of a module

1. define a type and its type bound procedures
2. define a collection of procedures within some category

3. define a collection of types and procedures sufficient for constructing a grammar and model of some domain

For modules in role 1, the module should have the same name as the type it defines. For modules in role 2, the module should have the name of the category, and for role 3, the module should have the name of the domain.

Derived types are generally used as a way to represent some quantity or entity. The name of a type should be a noun or noun phrase that describes the quantity or entity to be represented by the type.

Variables store a specific value or instance of a quantity or entity. The name of a variable should be a noun or noun phrase that describes the specific quantity or entity stored.

Procedures operate on and/or create quantities and/or entities. Procedure names should be a verb or verb phrase describing the operation performed, or the quantity/entity created.

4 Code Organization

4.1 Ordering

The ordering of executable statements is obviously important for the correctness of code, but there are other things for which the order does not matter to the compiler or for the correctness of the code, but a good convention can still make things easier to find. Also, the language defines that all `use` statements come before any declarations, and all declarations come before any executable statements. But there may be many of each of these statements, and they may contain many pieces. So let's talk about how to order them and their parts.

4.1.1 Use Statements

Alphabetize the `use` statements by module name. Always use the `only` clause to explicitly declare what you are importing from the module. Group the items imported by type(s), then procedure(s), then variable(s), and alphabetize within each group. Follow the last `use` statement with a blank line

4.1.2 Declarations

Declare one variable per statement. Group the declarations by argument(s), then parameter(s), then local variable(s), separated by blank lines. Declare the arguments in the same order that they appear in the argument list. Declare the parameters and local variables in alphabetic order. Follow the last declaration with a blank line.

4.1.3 Procedures

Order procedures alphabetically with blank lines separating them. There may be other, more logical orderings, but generally, the frequency that you need to

find a specific procedure far exceeds the frequency that you want to read them in a specific order.

4.1.4 Type Declarations

There are frequently inter-dependencies between types such that strictly alphabetic ordering is not possible, because a type must be declared before another type can have it as a component. Order type declarations as necessary to satisfy this requirement, but alphabetic otherwise, with blank lines between them.

4.2 Module Contents

A module should generally serve one of the following purposes: contain a single type declaration and its type bound procedures, contain a collection of related types, operators and procedures for a specific domain, or simply a collection of procedures within a specific category. The module should then be named for the type, domain or category respectively.

Additionally, to the extent that you can, you should limit the scope of any procedures. Any type bound procedures should be private, and only callable via instances of the type (i. e. `object%procedure(args)`). Procedures which are semantically the same, but with slightly different arguments, should be combined in an interface, with only the interface being made public. Any procedures which are not needed outside the module should be private. Any procedures which are used only from a single procedure should be contained in that procedure.

4.3 Files

A file should contain a single module or a single program, and should be named the same as the module or program. For small to medium sized projects, keeping all the source files in a single folder works just fine. When projects start to larger and more complex, a more elaborate organization becomes necessary. While Fortran does not support namespaces (yet), one could emulate the idea by simply using the following convention. Simply prefix module names with the namespace, and put all modules with a common namespace in a folder with that name. The file names then only need to include the last part of the module name. For example, a collection of operators for a vector type might be in a module `vector_operators_m`, in a file `operators_m.f90` in the folder `vector`.

5 Programming Constructs

These guidelines are intended to avoid common pitfalls leading to poorly designed or structured code. Such code is generally more difficult to understand and harder to debug. These guidelines are based heavily on standards developed by NASA [2] and JPL [4].

5.1 Simple Control Flow

All code should be restricted to relatively simple control flow constructs. Do NOT use `goto` statements, deeply nested branching/looping constructs, or indirect recursion. The use of recursion should be restricted to simple, commonly used algorithms.

Simple control flow translates into stronger capabilities for analysis, often resulting in improved code clarity. While this rule does not require all procedures to have a single point of return, this technique often simplifies control flow, except in situations where an early error return is the simpler solution. Procedures should only have one point of entry. Multiple points of entry are and indication that the procedure violates the single responsibility principle and should be separated.

5.2 Data Hiding

Declare all data objects at the smallest possible level of scope. No declaration in an inner scope shall hide a declaration in an outer scope.

This rule supports the well known principle of data hiding. If an object is not in scope, its value cannot be referenced or corrupted. Similarly, if an erroneous value of an object has to be diagnosed, it will be easier if there are fewer statements where the value could have been assigned. The rule discourages the reuse of variables for multiple, incompatible purposes, which complicates fault diagnosis.

The rule is consistent with the principle of preferring pure functions that do not touch global data, that avoid storing local state, and that do not modify data declared in the calling procedure indirectly. The use of distributed state information can significantly reduce code transparency, reduce the effectiveness of standard software test strategies, and complicate the debugging process if anomalies occur. Good programming practice is to prefer the use of immutable data objects and references. Procedure parameters should be declared with the qualifier `intent(in)` wherever possible.

5.3 Shared Data

Shared data objects should have a single owner, and only that owner should be able to modify the object.

Ownership equals write permission, but non ownership generally will not exclude read access to a shared object. Note that this rule does not prevent the use of system wide modules that are not associated with any one task or owner, but it does place a restriction on how tasks use such modules. Generally, if a shared object does not have a single owning task, access to that object has to be regulated with the use of locks or semaphores to avoid access conflicts that can lead to data corruption, and this process is highly error prone and likely to introduce bugs.

5.4 Language Compliance

All code shall be compiled with all compiler warnings enabled at the most pedantic setting available. All code must compile without warnings. The rule of zero warnings applies even when the compiler gives an erroneous warning: If the compiler gets confused, the code causing the confusion should be rewritten so that it becomes more clearly valid.

It is not uncommon for developers to get caught in the assumption that a warning is invalid, only to realize later that the message is in fact valid for less obvious reasons.

6 Comments

Comments are a contentious subject. Historically, the conventional wisdom has been that all code is hard to read and understand, therefore comments were seen as necessary for describing the what and how of a piece of code. The mantra "make sure you comment your code" has been seen as an unmitigated force for good, frequently known and repeated even by non programmers.

But, as we have seen throughout this guide, our goal is to write code that **is** easy to read and understand. If we have achieved that goal, then at best the comments are superfluous and redundant. Therefore we should view comments as an admission that we have failed to achieve our goal of properly expressing ourselves in the code. We should not view comments as necessary and a force for pure good. In fact there are reasons to avoid comments, as we shall soon see.

6.1 Comments are Noisy

When trying to read and understand a piece of code, every bit of text takes up some bandwidth of the information trying to get into your brain. If we have achieved our goal of well written code, then at best the comments are telling me things I already would have figured out from the code. Even worse, they may be irrelevant to the bit of logic I'm trying to understand. For example, reading a line like `j = j + 1 ! Increment i`, the comment is redundant, taking up mental bandwidth when I'm probably trying to figure what the loop that line is almost certainly within is doing.

6.2 Comments can Lie

The compiler doesn't check your comments. In fact, with the pace of development, when code is modified quickly under tight deadlines, trying to fix bugs and add features, the comments are almost certainly the last things to get updated, and are easily forgotten. Comments can then become out of date, irrelevant, or even outright wrong. For example, read the comment in the example in the previous section, and you'll notice that the comment says it increments `i`, but the code actually increments `j`. I've seen comments like that, and comments

that lie on an even grander scale, in quite sophisticated code bases. Having been bitten by this problem enough times, I am always quite skeptical when reading comments.

6.3 Comments can Float Away

Another thing that often happens to comments, is that they float away from the code that they actual try to describe. As code gets copied and pasted, code gets added in the middle, and other modifications happen, comments can end up quite far from the code they were written to describe. A reader can then easily be distracted from the task at hand trying to track down the code the comment is supposed to go with. These kind of stray comments can be at least as, if not more, noisy than redundant comments.

6.4 Commented Out Code

Don't leave commented out code lying around. Future readers will have no idea why it's there. Will something break if I uncomment it? Is something broken now because this is commented? Was this intended for some future development? You don't need to save it "for posterity". Your version control system will always have a copy if you ever really need it again.

6.5 When to Use Comments

Given all the reasons to avoid using comments, are there cases where comments are appropriate? Comments are most useful when they describe **why** a piece of code was written the way that it was, or how it is intended to be **used**. For example, a useful comment might be "this algorithm was chosen over x, y, or z algorithms because the type of data we're working with is more amenable to ...". This is the kind of information that would be almost impossible to express in the "how" of the code, but could be invaluable to future maintainers.

Another example of a useful comment might be "make sure the array is sorted before calling this procedure, because x, y or z might go wrong. Typical usage would be 'do_thing(sorted(the_array))'". There might be performance or organizational reasons that these types of assumptions or error conditions may not be expressed or checked for within the code, but would of course be important to keep in mind for any future maintainer.

Another acceptable use of comments is for use with automated documentation generators. It is frequently desirable to provide documentation for a library that describes the procedures and types that should be used by external code, and how they should be used. By keeping such documentation in the source code and using a tool to automatically extract it, it becomes at least somewhat more likely that it will be kept up to date as the code gets modified.

So while comments aren't all bad, they certainly aren't a force for pure good. They should be used judiciously and with purpose. What a piece of code does

and how it does it should be evident from the code itself. The why and use cases can, and often should be explained by some well written comments.

7 Additional Resources

For further information on coding conventions and style guides, I highly recommend reading the following items. The ideas and conventions expressed in this guide were heavily influenced by them.

- The Power of 10 [2]
- JPL Institutional Coding Standard [4]
- Pep 8 [7]
- Google Style Guide [3]
- Clean Code [6]

References

[1] Robert Bringhurst. *The elements of typographic style*. Hartley & Marks, Point Roberts, WA, 1996.

[2] J. Gerald. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–97, jun 2006.

[3] Google. Google c++ style guide.

[4] JPL. Jpl institutional coding standard for the c programming language. Technical Report JPL DOCID D-60411, Jet Propulsion Laboratory.

[5] Jonathan Ling and Paul van Schaik. The influence of font type and line length on visual search and information retrieval in web pages. *International Journal of Human-Computer Studies*, 64(5):395–404, may 2006.

[6] Robert C. Martin. *Clean Code*. Prentice Hall, 2009.

[7] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8.