Fortran Coding Standards and Style

The Fortran Company

Version 20160112

The source file for this document in Open Office format may be downloaded from

http://www.fortran.com/Fortran_Style.odt

This is intended to be an evolving document. Please send comments and suggestions to info@fortran.com.

1 Introduction

The following coding standards and style suggestions have been put together with the goal of making it easier to debug and maintain Fortran code.

Some the recommendations have arbitrary components, such as the number of spaces to indent in a block. The main thing is to pick a consistent style for a program, a project, or an organization. Others are recommendations that should definitely be followed whenever creating new code (e. g., put procedures in a module).

Nobody is going to agree to all of these suggestions. A good use of this document might be to serve as a basis for a customized style, obtained by doing some deleting, changing, and adding.

1.1 Program Structure

1. Programs should consist of a main program, modules, and submodules. Module procedures may be contained in libraries. All procedures should be internal or in a module or submodule.
2. Procedures should be small, typically 50 lines or less.
3. Derived-type definitions should be in a module.
4. Use a function if the procedure returns a single value (including possibly an array or structure) and the procedure has no side effects; otherwise, use a subroutine.

1.1.1 Main Program

1. The main program should be used only to call one or more high-level procedures.

1.1.2 Modules

1. Each module should be contained in its own source file, except that closely related modules might be put in the same file.
2. With the following two exceptions, the default accessibility of each module should be private by including the `private` statement:
   - A module containing only parameters
   - A "collector" module containing only `use` statements that access other modules

2 Declarations

2.1 Implicit None

All program units should contain `implicit none` so that all variables, parameters, and functions must be declared.

2.2 Form of Declarations

1. Pick a consistent style from among the many equivalent declaration statements. The simplest rule is to put all characteristics of an entity to the left of a double colon (`::`) and a list of names (possibly with initialization) to the right.
2. Use optional keywords to provide readability, for example

```
real(kind=dp), dimension(:, :) :: x_array, y_array
```

3. Put the declaration of each entity on a separate line, unless they are closely related, so that changing the declaration for one should also change the others.

```
character(len=MAX_CHAR_LEN), dimension(N, N) :: char_A, char_B
```

2.3 Names

1. Use short names for entities whose use is clear. For example, `i` and `j` are acceptable for array subscripts and `x` and `y` are fine as function dummy arguments. Use longer names for entities whose purpose is not obvious: `distance_to_the_moon`. Use different and consistent schemes for keywords, variables, parameters, procedures, modules, and types. The following is just one possible scheme. Some prefer to use lower case for everything.

Keywords	`print, select case`
Variables	`time, distance_To_The_Moon`
Parameters	`PI, GOLDEN_RATIO`
Procedures	`Distance, Number_of_Points`
Modules	`Point_Mod, Reactor_Simulation_Mod`
Type	`Point_Type, Reactor_Type`

2. Do not use the same name for two different things in the same scope, even when allowed. For example, do not use `print` as a variable name.
3. Do not use the same name with different uppercase/lowercase spelling, e. g., `N` and `n`.
4. Use the letters O and l and the digits 0 and 1 in names only when it will be perfectly clear which character is meant.
5. Each file name should be the name of the containing entity with the `.f90` extension. For example, the module `Point_Mod` should be in the file `Point_Mod.f90`.

2.4 Arrays

1. Index arrays with meaningful bounds. For example, an array containing the population of a city for the years 1900 to 2015 can be declared as

```
integer, dimension(1900:2015) :: population_Of_Tucson
```

3 Program Form

1. Use free-form source format.
2. Limit the line length to 80 characters.
3. Break lines in logical places. Indent continued lines double the number of spaces that each block is indented (see #6 below).
4. Insert meaningful comments frequently.
5. Use blank lines to separate related parts of a program.
6. Indent blocks a consistent number of spaces (such as 2, 3, or 4).
7. Use white space freely in statements (e. g., around delimiters).

4 Statements

4.1 Control of Flow

1. Use only the `block`, `if`, `do`, and `select case` constructs to control flow.
2. Labels should not be used except for a very rare `go to` / `continue` pair (see #3 and 4.2 below).
3. No form of `go to` (computed, assigned, etc.) should be used, except very rarely (e. g., exception handling?) a `go to` statement may branch *forward* to a labeled `continue` statement.

4.2 Format

1. Use a character string for each format. If it is used once, the string may appear in the data transfer statement itself.

   ```
   print "(a, f0.2)", "The price is $", price
   ```

   If it is used multiple times and does not change, use a character parameter; otherwise, a character variable.

   ```
   character(len = *), parameter :: PRICE_FORMAT = "(a, f0.2)"
       . . .
   print PRICE_FORMAT, "The price is $", price
   ```

5 Miscellaneous

1. Do not use nonstandard or obsolescent features.
2. Do not write "tricky" code to save a few lines or a few milliseconds. If execution efficiency is important, write code that reflects the algorithm, instrument the code, try some variations, and comment liberally including the original code.
3. Do frequent error checking. For example, test if dummy argument values are reasonable, use status variables for input/output and allocation, and use the `inquire` statement prior to opening a file. Always test a status variable after it is used. If the problem is not clear, try removing the status check and see what the system produces.
4. Use parameters (named constants) instead of integer, real, or complex literal constants.
5. All character dummy arguments and parameters should be declared with length `*`.
6. All functions should be pure and have no side effects (note: what constitutes a side effect is controversial). This restriction may be removed temporarily to use debugging output statements, if a graphical debugger is not being used.
7. All array dummy arguments should be assumed shape (`:`).
8. All dummy arguments should have their intent declared. All function dummy arguments should be intent `in`.
9. Use keyword argument calls frequently, including with intrinsic procedures.
10. Use keywords frequently in structure constructors.
11. Each `use` statement should have an `only` clause to identify the source of the used entities. An exception might be for a module with a very large number of used entities.
12. When both pointers and allocatables will do the job, use allocatables.
13. Use the NEWUNIT intrinsic to select am input/output unit number. Never use single-digit unit numbers for input/ouput.
14. For standard and error input/output units, use `INPUT_UNIT`, `OUTPUT_UNIT`, and `ERROR_UNIT` from the intrinsic module `ISO_FORTRAN_ENV`.
15. Do not check for equality or inequality between real or complex values; use "closeness" instead.
16. Use logical variables for flags, not integers (and certainly not reals).

17. In a context that requires conversion of complex to real or integer or conversion of real to integer, use the intrinsic conversion functions `real` or `int`.

18. Put at least one digit on each side of the decimal point in real and complex literal constants.

Index

S

U

V

W