Fortran Coding Standards

Best Practices Workshop, March 25-26 2019, Hampton VA

Carlos Cruz
Jules Kouatchou
Brent Smith

NASA GSFC Code 606/610 (ASTG/GMAO)
Greenbelt, Maryland 20771

Quotes

> *Establish programming conventions before you begin programming. It's nearly impossible to change code to match them later.*

Steve McConnell, Code Complete (Second ed.). Microsoft Press, 2004.

> *Programs must be written for people to read, and only incidentally for machines to execute.*

Abelson & Sussman, Structure and Interpretation of Computer Programs

Purposes

Provide guidance for:

- The selection of names, formatting of structures,
- The use of comments and other issues

Why Use Conventions

- Important when a project involves more than one programmer.
- Much easier for a programmer to read code written by someone else if all code follows the same conventions.
- Write readable codes.
- Write maintainable codes.
- Help to write clear, accurate and precise user documents.

When NOT To Use These Conventions

1. Customer's preferences
2. Existing codes
3. Extending a framework

When To Use These Conventions

1. New Files
2. Simple changes

Naming Files

- File names shall generally use the same name as the class/module they implement.
- Fortran filenames shall include the _mod suffix from the module.
- Fortran header files end with ”.h”.
- Fortran source files end with ”.F90”.

```
someModule_mod.F90
definedConstants.h
```

File Organization

Fortran files shall contain the elements in this order:

- **program, module, procedure, function**
- **use module** statements.
- **implicit none** declaration
- **private** (as default; **public** entities are declared explicitly)
- Include files
- Variable declarations (dummy arguments may appear before includes, then locals)
- Source code
- **contains** block

Expressiveness and Scope

- Identifiers with larger scope shall have more expressive names since they are useable in a larger body of code.
- Using i, j, k for temporary variables in **do** loops is generally acceptable when the loop is not long

```
subroutine recordDatabase (i, currentRec )
     integer , intent (in) :: i ! large scope , not acceptable !
     integer , intent ( out) :: currentRec ! much better

```

Abbreviatons

- Acronyms should be avoided if at all possible.
- Use all upper case letters for the acronym. Put underscores between the acronym and other capital letters.
- If the identifier needs to start with a lower case letter, such as in a variable name, then use all lower case letters for the acronym. Do not use an underscore after the acronym.

Case and Underscores

- Identifiers (and keywords) should be named consistently across programs and among developers.
- Underscores should be used only when necessary, such as in the use of all capital letters in parameters (MAX_NAME_LENGTH) or when the term may become unclear.
- Optional parameters with default values shall use an underscore at the end to differentiate the local variable used to assign it a value.

```
subroutine foo ( someValue )
    integer , optional , intent ( inout ) :: someValue
    integer :: someValue_

    someValue_ = defaultValue
    if ( present ( someValue )) someValue_ = someValue
    ...
end subroutine foo

```

Routines

- Routines shall begin with a verb, preferably a strong action verb.
- Accessor and mutator functions shall begin with **get**, **set**, and **is**.
- Avoid global routines and place routines inside of a related class or module to avoid naming collisions.

```
subroutine parseMessage ( inputMessage )

function getSurfaceArea(shape_object)

```

Variables and Arguments

- Use descriptive variable names
- Variables and arguments shall be named with nouns since they represent a thing or quantity.
- All variables must be explicitly initialized before use, avoiding problems with the assumed value of uninitialized variables.
- Multiple declarations per line shall be avoided, unless the variables are very tightly coupled.
- Constants shall be used instead of literal constants (magic numbers). item Boolean values shall be used rather than 0 or 1.

Argument Modifiers

The *intent* of each argument (i.e. *in, out, inout*) shall be specified before each argument declaration in a routine.

```fortran
subroutine updateSurfaceAreaDensity ( initNum , finalCond )
    implicit none
    integer , intent (in ) :: initNum
    real *8, intent ( out ) :: finalCond

    ! ...

end subroutine updateSurfaceAreaDensity
```

Implicit None

- **implicit none** shall be at the top of all program units to ensure that variables are explicitly declared, documented, and type checked.
- It is the default in module functions if declared at the top of the module.

Automate I/O Unit Numbers

- An inline comment is a comment on the same line as a statement.
- Inline comments should be separated by at least two spaces from the statement.
- They should start with a # and a single space

```
subroutine updateSurfaceAreaDensity ( initNum , finalCond )
    implicit none
    integer , intent (in ) :: initNum
    real *8, intent ( out ) :: finalCond

    ! ...

end subroutine updateSurfaceAreaDensity
```

Documentation String

- As of Fortran 2008, the language provides a **newunit** specifier to the open statement, which shall be used to obtain I/O unit numbers
- This **newunit** intrinsic automatically assigns a unique negative unit number, preventing conflicts with any existing unit numbers.
- No need to hard code constants such as the numbers 5 and 6.

```fortran
open ( newunit = myUnit , file ='surface_data.txt ', ...)
read ( unit = myUnit , iostat = ioerr ) sfcTemp
```

Class and Module Identifiers

Class and module modifiers, like **public**, **private** and **implicit none**, shall be indented, with the exception of **contains**.

```
module BankTransaction_mod
     public creditAccount
     implicit none

contains
     subroutine creditAccount ( account )
          ! ...
     end subroutine creditAccount
end module BankTransaction_mod
```

Derived Types

The following naming conventions shall be used for derived type constructors and destructors:

```
Type name:           Foo
Module name:       [package_name_]Foo_mod
File name:            [package_name_]Foo_mod.F90
Constructor interface name: Foo
Constructor name:      newFoo
Destructor name:       destroyFoo
```

Class Names

```fortran
module BankTransaction_mod
      public creditAccount
      implicit none

contains
      subroutine creditAccount ( account )
            ! ...
      end subroutine creditAccount
end module BankTransaction_mod
```

Function and Variable Names

- Function names should be lowercase, with words separated by underscores as necessary to improve readability.
- Variable names follow the same convention as function names.

Sample Code

```fortran
module Foo_mod
      ...
      type Foo
            integer :: windDirection
            real :: windSpeed
      end type Foo

      interface Foo
            module procedure newFoo
      end interface

contains

      function newFoo () result ( this )
            type ( Foo ) :: this
            this%windDirection = 0
            this%windSpeed = 0
      end function newFoo
      ...
end module Foo_mod
```

Use-Module Items

- When introducing items from a module, it is preferable to explicitly identify the entity you want to use.
- This helps to easily identify the origin of the item.

```fortran
use Foo_mod , only : startBar1 , stopBar ! good , identifies methods precisely
use Foo_mod ! bad , where do Bar methods come from ?

```

Statements

- One statement per line.
- One operation per statement

Pure Block Layout

- Pure block layout shall be used to layout a block of statements.
- Each sub-block shall be indented another level.

```fortran
do i = 1, 100
     statement1
     statement2
     if ( statement3 ) then
          print *, statement4
     end
end do
```

Named Blocks

- Labels should be used for longer blocks of code to provide clarity, especially if there are multiple inner loops.
- Labels also provide an elegant method to exit an outer block from the middle of an inner block.

```
Outer : block
     InnerLoop : do i = 1, 5
          ...
          if (x > X_MAX ) exit Outer
          ...
     end do InnerLoop
     call someRoutine
```