# ATM 562 – Some Fortran programming pointers

Fall, 2018 – Fovell

Some quick pointers on Fortran programming and program structure. I make no claim to being a master programmer (far from it!), but I think these suggestions will help make your program easier to write, to read... and therefore, to debug. These pointers are mainly based on Fortran 77 (F77), and emphasize clarity and readability over "style" and efficiency. Some initial points:

- Fortran is NOT case sensitive.
- Variables are assumed real, unless the first letter of the variable name starts with the letters I-N inclusive. Then they're integers by default. (Mnemonic device: I-N are INtegers.)
- Array indices must be integers, so use index variable names like "i", "j", and "k".
- Comment lines start with "C" or "c" in the first column, or "!" anywhere, and comments can be appended to statements following the "!" character.
- Fortran statements start in column 7. Column 6 is for continuation markers. Columns 1-5 are for statement labels.

A sample program:

```
      program example       ! this is optional, actually

! non-executable statements go first: this includes parameter, common,
!  dimension, data statements and statement functions

! parameter statements make array bookkeeping easy
      parameter(nx=100,nz=40)
! common blocks hold arrays, constants you want to use in
!  multiple program modules
      common/base/uinit(nz),tinit(nz),qinit(nz),pk(nz),rhou(nz),rhow(nz)
      common/uwind/up(nx,nz),u(nx,nz),um(nx,nz) ! horiz wind at 3 times
      common/ptemp/thp(nx,nz),th(nx,nz),thm(nx,nz) ! theta pert at 3 times
      common/grids/dx,dz,dt,d2t,time   ! grid setup, etc
      common/grid2/zw(nz),zu(nz)  ! height of W and U points on grid
      common/consts/rd,g,cp,psl     ! physical constants
! dimension statements for local arrays - used only in this program unit
      dimension crap(nx,nz)
! initialize constant values you will not change during execution
      data g/9.8/,dx/400./,dz/400./,dt/2.0/,cp/1004./,rd/287./
```

```
!-----------------------------------------------------------------------
! executable statements go here
!-----------------------------------------------------------------------

! a neat way of getting trigonometric ‘‘PI’’ to machine precision
      trigpi=4.0*atan(1.0)

! frequent divisions should be converted to multiplications where possible
!  ex: I need to divide rd by cp a lot, and also (dx**2) and (dz**2)
      xk=rd/cp
      rdx2=1./(dx*dx)
      rdz2=1./(dz**2)

! block DOs are easy to read when indented.
! every DO has an ENDDO
! float turns an integer into a real before multiplication

      do k=2,nz
       zw=(float(k-1)*dz)
      enddo

! block IFs may be simple or compound

      iflag=0
      do k=2,nz-1
       ztp = (float(k)-1.5)*dz
       if(k.eq.2) iflag=1

       if(ztp.gt.ztr)then
        tinit(k)=thetatr*exp(9.8*(ztp-ztr)/(cp*ttr)
       else
        tinit(k)=300.+43.*(ztp/ztr)**1.25
       endif

       if(k.le.20)then
         [do something here]
       else if(k.lt.nz-5)then
         [do something here]
```

```
       else
         [do something here]
       endif
      enddo


! continuation marker is most any symbol placed in column 6
!   for multiline statements

! **line up** your statements for easy debugging and reading

      do k=2,nz-1
       do i=2,nx-1
        thp(i,k)=thm(i,k)
     1      -0.5*dtx*(u(i+1,k)*(th(i+1,k)+th(i,k))
     2               -u( i ,k)*(th(i-1,k)+th(i,k)))
       enddo
      enddo

! subroutine calls make program neater, more modular
!  try to avoid passing unnecessary arguments -- use common statements

      call setup
      call runmodel
      call cleanup

! write statement: unit 6 is the screen
!  here: a real is written with format F5.2, an integer with I6,
!   and a real in exponential notation with E15.6)

      write(6,1000) real,integer,real
1000  format(1x,f5.2,i6,e15.6)

! you can avoid format statements when you don't need them

      write(6,*) ' the real number is ',real

! write to the screen with print, which is the same as write(6,*)

      print *,' the real number is ',real
```

```
      print 1000, real, integer, real

! open a file before you write to it
!  this opens a NEW formatted file called output at unit number 12
!   if this file exists, it will cause an error
!  avoid unit numbers 5 and 6; those are standard input and output

      open(12,file='output',status='new',form='formatted')

! or open the file with status UNKNOWN which will overwrite any old
!   file of same name
      open(13,file='output2',status='unknown',form='formatted')

! write to the file using the designated unit number
      write(12,*) rd,g,cp

! close the file when you are done
      close(12)
      close(13)

! program ends with ``stop'' and ``end'' statements
      stop
      end

! then your subroutines go here.  they terminate with ``return'' and
!  ``end'' statements

      subroutine setup

      [your parameter and common blocks repeated here]

      return
      end
```

Using global include files presents both a great bookkeeping advantage and a problem. Since you have to use the same parameter, common statements in the main program and each applicable subroutine, if you change these statements in any program unit, you have to make sure they're also altered in all other units. (Example, you change NX; need to do that in all program modules.) Failing to do this can cause errors that can be hard to track down.

Alternative is to use an external file called "global" or "storage" or similar. The name does not matter. Define your parameter and common statements (but not data statements, typically) in the global file, and then use INCLUDE to insert them into your program modules at the appropriate places. For example:

```
      program example2
      include 'global'

      [etc.]
      stop
      end

      subroutine setup
      include 'global'

      [etc.]
      return
      end
```

The potential difficulty with this is if you change your `global` file, you need to recompile the program. If you use a Makefile to compile your program, the compiler may not realize the global file has changed. So, to work around this, whenever you alter the contents of the global file, then use the Unix command "touch" to update the date/time associated with the fortran program (type `touch yourprogram.f` at the command prompt).

Recent versions of Fortran permit arrays to be manipulated "as a whole", so long as computations are done on arrays sharing the same shape. Example:

```
      parameter(nx=50,nz=40)
      dimension pi(nx,nz),temp(nx,nz),theta(nx,nz)

! This
      do k=1,nz
       do i=1,nx
         temp(i,k)=theta(i,k)*pi(i,k)
       enddo
      enddo

! is equivalent to this
      temp=theta*pi
```

One avoidable problem is the mixing of real and integers in arithmetic expressions, as the results may differ from expected depending on the order of operations. You can convert reals to integers using `ifix` and integers to reals using `float`.

A great disadvantage of F77 was the need to identify continuation markers in column 6 (making columns 1-5 unavailable to your statement) and the 72 column limit. Modern Fortran (i.e., Fortran 90/95, or F90) is not so limited. Continuation markers in F90 use the ampersand, as illustrated in this poor (because it could be very easily misread) example:

```
! Fortran 77 style (file name using .f suffix usually defaults to F77)
      trigpi2 =  4.0*atan(1.0)
     1                /2

! Fortran 90+ style (file name using .f90 suffix indicates F90)
      trigpi2 =  4.0*atan(1.0) &
                       /2

! ...which is the same as
      trigpi2 =  4.0*atan(1.0) &
              &        /2
```

Two ampersands can be used to mark continued statements, but one suffices.

Here are some compiler options specific to `gfortran`.

- `-fbounds-check` checks on array boundaries during execution. Really slows down execution, so use for debugging.
- `-ffpe-trap=zero` will cause program to terminate if there is a divide by zero.
- `-ffpe-trap=overflow` will cause program to terminate if there is an overflow.
- `-g` enables extra debugging information. `-g3` enables even more debugging information.
- `-Wall -Wextra` asks compiler to warn about all detected issues.
- `-O2` requests optimization level 2. `-O0` (no optimization) is default. `-O3` is aggressive.