Supplementary Notes for An Introduction to Fortran Programming

April 27 2007

These notes are a brief supplement to the references provided in the documentation section below.

1. Documentation

- A Simple Fortran Primer – Rosemary Mardling (1997) (primer.pdf)
  Important note: This primer tends to use `print *,` rather than the common `write (*,*)`

- Professional Programmer's Guide to Fortran 77 – Clive Page (1995) (reference.pdf)

- On atlas: `man f77` . On atlas and the Linux machines: `man g77`, `g77 --help`, `info g77`

- Sun Fortran manuals on shelf in Unix Lab (Room 408)

Note: PDF files can be read with Acrobat Reader (`acroread` on atlas or the Linux machines).

2. Program source file (`.f` file)

For historical reasons dating back to the days of punched cards the standard Fortran statement record or line is 72 columns (characters) wide:

```
------------------...--------
1    567                    72
↑          ←  Statement (code) in columns. 7-72 →
Column
```

Long statements (exceeding column 72) can be continued on the next line by putting a character in column 6 (often a *). Statement labels e.g. on `format` statements are from 1-99999 in columns 1-5 (any justification). Comments have a c in column 1. You can also use ! at the end of a statement

e.g. `100□□□format('x= ',F8.2)□□!A comment` (□ is a blank space)

Code from other source files can be incorporated with the `include` statement e.g. `include 'extracode.f'`. When the `f77` or `g77` command is used then the code in the file `extracode.f` will be 'pasted' into the current file at the `include` statement. We use this approach in Lab Session notes example 11 (c) (`corr3.f`).

3. Assignment of variables

As a general rule you will have the statement:

```
implicit none
```

as the second line of your program. This means that you must explicitly declare all variables in the same way as the C programming language. Without this statement it is assumed that variables beginning with I-N are integer e.g. `icount` and A-H and O-Z are real e.g. `radius`.
The = sign is interpreted as 'assigned to'. Each variable or array element e.g. `x`, `p(3)` has a memory location allocated to it. Assuming that `x` is declared as real then the statement `x = 2` stores the real number 2.0 in the memory location allocated (or reserved for) the variable `x`. If we have the two statements:

```
x = 2
x = x + 5.7
```

then the second statement takes the current value of `x` (2.0), adds 5.7 to `x` to get 7.7 and then *overwrites* the memory location for `x` with the new value i.e. after the second statement `x` is now 7.7.

4. Format descriptors

In the 'old days' numbers had to be right-justified for `read`. This doesn't seem to apply anymore. In the following the field width `w` includes the decimal point and +/– . `d` is the number of decimal places. Note that for `write` numbers are output as right-justified and text as left-justified.

(1) Real or floating point numbers

```
                ←d→
Fw.d      ----•--        e.g. F7.2
          ←  w  →
```

e.g. `a = 104.5627` will be output as □`104.56` (note rounding to fit format)

(2) Exponential representation of floating point numbers

```
                ← d  →
Ew.d      ±0•------E±xy          e.g. E13.6
          ←      w     →
```

This always begins with a decimal part less than one.
e.g. `a = 104.5627` will be output as `□0.104563E+03` (note rounding to fit format)

(3) Integers

```
Iw        ------          e.g. I6
          ←  w →
```

e.g. `j = 4` will be output as `□□□□□4`

(4) Character or text variables

```
Aw   ------          e.g. A6
     ←  w →
```

Character variables are written left-justified
e.g. `label= 'date'`
With the above format (`A6`) this will appear as:
`date□□` (□ is a blank)

5. Input

(1) Read a real number from the keyboard and assign to real variable `x`.

```
real x
read(*,*)x
```

keyboard ↑ ↑ free format

(2) Read an integer from the keyboard into variable `i`.

```
integer i
read(*,*)i
```

(3) Read in 5 real numbers from the keyboard into array `a`.

```
integer i
real a(5)
read(*,*)(a(i),i=1,5)        ← implied DO loop
```

The last line is equivalent to: `read(*,*)a(1),a(2),a(3),a(4),a(5)`

(4) In many cases we read data from a file and we're not sure how many numbers there will be in the file. For simplicity we will assume that the file press.dat contains a column of numbers e.g.

1021.1
1000.3
1001.7
…

```
integer i,n
real p(100)
open (1,file='press.dat') ← Unit 1 is assigned to the file press.dat (a kind
                            of short name or handle)
do i=1,100
  read(1,*,end=9)p(i)
enddo
9 continue  ← Go to here if we encounter the end of the file (end=9)
n= i -1     ← Last value of i corresponds to the end of the file (one greater)
write(*,*)'n=',n
```

We can now process the array of numbers p(i) i=1 … n .

(5) In (4) it was assumed that we were reading in a single column of numbers. Imagine that the file press.dat contains the following:

```
Data□□□□□Pressure      ← □ denotes a blank space
850601□□□1021.1
850602□□□1000.3
850603□□□1001.7
↑
Column 1
```

To read in the second column of numbers into array p:

```
character line*80          ← A variable to hold 80 characters
integer i,n
real p(100)
open (1,file='press.dat')
read(1,*)line or read(1,'(A)')line
do i=1,100
  read(1,'(9x,F6.1)',end=9)p(i)  ←Formatted read statement
enddo
9 continue
n= i -1
write(*,*)'n=',n
```

We also use a format statement:

```
read(1,200,end=9)p(i) ← Use the format statement with label 200
200 format(9x,F6.1)
```

which is equivalent to:

```
      read(1,'(9x,F6.1)',end=9)p(i)
```

(6) To read in the line of text as two column labels and both columns of data:

```
      character label1*4, label2*8   ← Variables to hold column labels of 4 and 8 characters
      integer i,n
      real p(100)
      integer date(100)        ← Array to hold integer date
      open (1,file='press.dat')
      read(1,210)label1,label2
210   format(A4,5x,A8)  ← Format for labels
      do i=1,100
        read(1,'(I6,3x,F6.1)',end=9)date(i),p(i)  ← Formatted read statement to read date(i) and p(i)
      enddo
    9 continue
      n= i -1
      write(*,*)'n=',n
```

6. Output

Generally `write` is just the opposite of `read`.

(1) Write a real number `x` to the screen.

```
      real x
      write(*,*)x
          screen ↑  ↑ free format
```

In the primer: `write(*,*)x` is equivalent to `print *,x`

(2) You can use a `format` statement and include some output text:

```
      write(*,100)x
100   format('The answer is ',F8.2)  equivalent to
      write(*,'(''The answer is '',F8.2)')x   ← Note repeated quotes
```

7. Internal `read` and `write`

This involves reading from or writing to a character variable that is treated as a 'pretend' file. For instance:

```
integer n
character*80 optarg   ← or character optarg*80
read(optarg,*)n
```

The integer variable `n` is read from the character variable `optarg` as if it were a line of text in a file. Similarly for a real variable we can have:

```
read(optarg,*)x
```

We can also use a format e.g. `read(optarg,'(2x,F6.1)')x`
Similarly we can write to a character variable:

```
write(optarg,*)x
write(optarg,'(2x,F6.1)')x
```

8. Logical variables

See the primer and reference. These are declared via:

```
logical lexist
```

A logical variable is either true (1) or false (0). Usually we use an `if` statement with a logical expression involving our variables e.g.

```
if (x.eq.5) then  ← Is x=5? If so then go to the then section otherwise go to the else section
  y = 2           ← If x=5 then go here and set y=2
else
  y = -4          ← If x is not 5 then go here and set y=-4
endif
write(*,*)'x= ',x,' y= ',y
```

If the expression in `(…)` is true the `then` section is executed otherwise we jump to the `else` section. Sometimes we code the above as:

```
logical istrue
istrue= (x.eq.5)  ← This will be either true (1) or false (0)
if(istrue)then    ← If x is 5 then istrue will be 1 and we go to the then section
...
endif
write(*,*)'istrue=',istrue,' x= ',x,' y= ',y
```

Note: If you write out a logical variable you will see `T` for true (1) and `F` for false (0) on most systems.

9. Mixed mode arithmetic

An expression that involves a combination of real and integer variables will give a real value, as long as the output variable is real.

```
real x,y
integer i,j
y= x/i          ← y is real
j= x/i          ← j is an integer i.e. integer truncation of x/i
```

Hence if `i= 4` and `x= 11.` then `y= 2.75` and `j= 2` i.e. 2.75 is truncated to 2.

An expression involving integers only will be the integer truncation.

```
integer i,j,k
real y
k= i/j          ← k is an integer i.e. integer truncation of i/j
y= i/j          ← y is the real conversion of the integer truncation
```

Hence if `i= 7` and `j= 2` then `k= 3` (integer truncation 7/2= 3).
Note that `y= 3.` i.e. the integer 3 is converted to real (3.0).

10. do loops

These are covered in the primer and reference but a few comments will be made. `do` loops are convenient for summations used to compute the mean and other statistics. Consider the mean of the sample $x_1, x_2, \ldots, x_n$:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The summation is $x_1 + x_2 + \ldots + x_n$. If we identify an array `x` of size `n` with the sample then the summation is `x(1) + x(2) + ...+x(n)`. Alternatively the summation can be expressed as a set of successive accumulations:

```
s1 = x(1)
s2 = x(1) + x(2) = s1 + x(2)
s3 = x(1) + x(2) + x(3) = s2 + x(3)
...
```

We can use a do loop to represent these accumulations:

```
xbar = 0.                ← Initialise xbar i.e. start with a sum of zero
do i=1,n
  xbar = xbar + x(i)     ← Add x(i) to previous sum to get current sum
enddo
xbar = xbar / n          ← Get the mean by dividing the sum by n
```

In older code the form:

```
     do 200 i=1,n
          ...
200 continue
```

is often seen.

11. Binary (unformatted) files

The files encountered in the primer and reference are formatted files that are appropriate for *text* data. We open such files with a statement like:

```
open (1,file='press.dat')
read (1,*)x
```

Much of the output from GCMs and related reanalysis projects e.g. NCEP is in a *binary* form i.e. not readable to the naked eye. Such files are opened and read in a different way. Assume we have a binary version of press.dat called pressb.dat:

```
open (1,file='pressb.dat',format='unformatted')
read (1)x
```

In the `open` statement we need the extra keyword `format` set to `'unformatted'` that means binary in Fortran. Also in the `read` statement we just have the unit number without any format

12. conmap (CMP) files

We have software that takes the binary files from (say) NCEP and converts them to a standard binary format which we call *conmap* (CMP) format. This is capable of representing a variable e.g. pressure, on a longitude-latitude grid.
A fragment of code to read a CMP file) is:

```
implicit none
real xmiss
parameter (xmiss= 99999.9)    ! conmap missing value code
integer num
parameter(num= 200)           ! Max. size of lat.-lon.
                              ! grid
real xlats(num),xlons(num)    ! Arrays for lats and lons
real dat(num,num)             ! Array for input data
character*80 head1            ! Header (description)
character*80 infile           ! CMP file name
open(1,file=infile,form='unformatted')
read(1)nlats
read(1)(xlats(i),i=1,nlats)
read(1)nlons
read(1)(xlons(i),i=1,nlons)
read(1)head1
read(1)((dat(i,j),i=1,nlons),j=1,nlats)!See conmap.f
close(1)
write(*,'('' No. lats, no. lons: '',2I6)')nlats,nlons
write(*,'(1X,A)')head1
```

We use a `parameter` statement to set the maximum size of our longitude and latitude arrays to 200 points (`num`). Thus our two-dimensional data array (matrix) `dat` is 200 x 200 points. The integers `nlons` and `nlats` hold the actual number of longitudes and latitudes e.g. `nlons = 144`, `nlats = 73`. The array `xlons` holds the actual longitude values e.g. 0.0, 2.5, … and the array `xlats` holds the

actual latitude values from south to north e.g. –90.0, -87.5, … The array `xlons` is read using an implied `do` loop i.e.
`read(1)(xlats(i),i=1,nlats)` is equivalent to
`read(1)xlats(1),xlats(2),…,xlats(nlats)`
The array `xlons` is read in a similar way. There is an 80 character text header (`head1`) containing some information about the data. Finally we read in the two-dimensional array `dat` using a pair of 'nested' implied `do` loops:

```
read(1)((dat(i,j),i=1,nlons),j=1,nlats)
                  ↑ i is longitude index and j is latitude index
```

This works from the 'outside' to the 'inside' loop. We start with `j = 1` and read the inner loop (`dat(i,1),i=1,nlons)`. Then `j = 2` and we read (`dat(i,2),i=1,nlons)` and so on. For each latitude index `j` we read the values of `dat` at each longitude index `i` i.e. we read the data on latitude circles from south to north. If we specified each array element individually we would need:

```
read(1)dat(1,1),dat(2,1),…,dat(nlons,1),dat(1,2),dat(2,2),…
dat(nlons,2),…,dat(1,nlats),dat(2,nlats),…dat(nlons,nlats)
```

To write out a CMP file we just replace `read` by `write` in the above code.
The conmap format has a code to represent missing or undefined data (99999.9) as set by `xmiss`. This is useful to process files with missing data since we want to exclude these from our summations etc. See: `statcon.f`.

13. Functions

These are a special kind of variable. There is no need to declare intrinsic (built-in) functions e.g. `sin` but you need to declare your own e.g. `probst` in the correlation examples. A function returns a value that is normally assigned to a variable e.g.

```
real x,y
y = sin(x)
```

Here the value of `x` is input to the intrinsic function `sin` which returns the value `sin(x)`, assigned to the variable `y`.

14. Subroutines

These don't require declaration. There is an *association* between the variables in the calling routine and the subroutine. Variables must *match* in terms of type e.g. real, and array size but they can have different names. For instance:

```
program test2
integer n
real x(100)
real meanx
call calcmean        (n,x,meanx)
…                     ↓ ↓ ↓
end
subroutine calcmean (n,y,ybar)
integer n
real y(n)
real ybar
…
return
end
```

The subroutine takes the array `x` of size `n` i.e. `x(1), x(2), … x(n)`, computes the mean `ybar` and passes this back with the name `meanx` in the calling routine. Note that `n` and `y` are also passed back so if they are modified in the subroutine then they will be modified in the calling program after the `call` statement. Usually we label subroutine arguments that are *not* modified as *inputs* and those that are as *outputs*.

15. Command-line arguments: getarg, iargc

These are useful intrinsic (built-in) routines to simply input and output.
Imagine we have a program called `jason1`. By using the above routines we can read filenames and other arguments from the command line. In our case we want to give an input file (argument 1) and an output file (argument 2)
e.g. `jason1 press.dat mean.dat`
↑argument 1 ↑argument 2

A program fragment to do this is:

```
character optarg*80, infile*80, outfile*80
integer narg
narg= iargc()      ← This will be 2 in the above example i.e. 2 arguments
write(*,*)'No. of arguments: ',narg
call getarg (1,optarg) ← Get first argument from command line and put into optarg
                ↑ Argument 1

infile= optarg ← This variable holds the name of the input file
call getarg (2,optarg) ← Get second argument from command line and put into optarg
                ↑ Argument 2
```

```
outfile= optarg ← This variable holds the name of the output file
open(1,file=infile)    ← Open the input file
open(2,file=outfile)   ← Open the output file
```

We can also read numerical arguments from the command line
e.g. `jason1 press.dat mean.dat 24.5`
↑argument 1 ↑argument 2 ↑argument 3

Argument 3 can be read by adding:

```
real x
...

call getarg (3,optarg) ← Get third argument from command line and put into optarg
                ↑ Argument 3

read(optarg,*)x    ← Do an internal read from optarg i.e. read x from optarg
write(*,*)'x= ',x
```

This approach is very useful for handling a large set of input files
e.g. `jason1 press.9601?? mean.dat`

Under UNIX or Linux the argument press.9601?? is expanded into a set of filenames which differ in the last two characters. Note that at least one of these files must exist. Assume that the files are called press.960101, press.960102, press.960103, …, press.960107 i.e. daily pressure files for Jan 1 1996 to Jan 7 1996. Then the command line will be expanded as if the individual files were given i.e. as if we had specified:

```
jason1 press.960101 press.960102 … press.960107 mean.dat
         ↑ Argument 1        ↑ Argument 2          ↑ Argument 7        ↑ Argument 8
```

Such an approach is used in `statcon.f` (see Lab Session notes example 9).

16. Common blocks

We won't discuss these in detail.
See Lab Session notes examples 10 (`cblk1.f`, `cblk2.f`) and 11 (c) (`corr3.f`).

17. Makefiles

We won't discuss these in detail but see example 11 (d) (`corr4.f`) in the Lab Session notes.

18. Error messages

In general, Sun `f77` compilation and runtime messages are often cryptic. The messages from `g77` tend to be clearer. The following are some common errors encountered using Sun `f77`. They have a similar expression using `g77`.

- *list io: [-1] end of file*
  logical unit 1, named 'press.dat'
  part of last data 11.0 ^J |
  Abort
  This occurs if you try to read past the end of a file i.e. the program expects more data than is given in the file.

- *Killed*
  There is not enough system memory to run the program. This could be due to the presence of other processes on the system consuming most of the available memory or the program's memory requirements exceed the limits for the system.

- *Segmentation fault*
  This usually means that the memory allocated to the program as it is running has been corrupted in some way. It often arises with illegal array processing e.g. try to write out `a(1)` … `a(1000)` when the maximum size of `a` is (say) 5.

- *I/O error*
  This occurs when you have the wrong format e.g. `read (1,'(I5)')x` where `x` is real. Note that this is a runtime error – depending on the compiler it may not show up in the compilation stage.

- *Subscript out of bounds*
  This occurs when you try to process an array element outside the valid range when the program was compiled with `f77 -C` (Sun) or
  `g77 -ffortran-bounds-check` (Sun or Linux PC)
  e.g.
  ```
  real a(5)
     write(*,*)a(6)
  ```
  The element `a(6)` is not valid so using the above options will cause the program to abort.