Lab Session for An Introduction to Fortran Programming

March 9 2010

Source code and data files

1. Log on to a Linux machine e.g. `abyss`. You should be in your home directory i.e. `/home/user`
2. `mkdir fortran`
3. `cd fortran`
4. `cp /home/keay/fortran_course/zfortran_course.2007.zip .`
5. `unzip zfortran_course.2007.zip`

You will see a collection of files and a subdirectory (folder) called corr4.dir.
There is no need to run all of the examples in the lab session. You can run them at your leisure.

To compile, use `f77` or `g77`. Note: On atlas (SUN) `f77` and `g77` are *different* compilers while on the Linux machines e.g. cove, they are *equivalent*. In these notes we will mainly use `g77` since it is available on many common platforms. Some reference is made to NCAR Graphics. There is a manual in the UNIX Lab – it is just below the Safety Manual, with the Fortran reference. Some modifications to your initialisation files i.e. .login, .cshrc, are required for Linux – these should have occurred during the *Introduction to UNIX* course.

Examples

Example 1
Type in the following short Fortran program (source code). The ‘ruler’ at the top indicates the column number (don’t type this line). Comments have a c in column 1 and statements begin in column 7 or greater with the last column being 72. Use a character e.g. * in column 6 for long statements. As a general rule, *spaces and case don’t matte*r.

`1234567` ← **Don’t type this line!**

```
      program first
c
      implicit none
      real x,y
c
      write(*,*)'My first program'
      write(*,*)'Enter a number: '
      read(*,*)x

      y= x**2 +4*x -12

c * Free format output
      write(*,*)'x=',x,' y= ',y

c * User-specified formatted output
      write(*,10)x,y
10    format('x=',F8.4,2X,'y=',F9.5)
      end
```

To compile, use `f77` or `g77`. Note: On atlas `f77` and `g77` are different compilers while on the Linux machines e.g. cove, they are equivalent. In these notes we will mainly use `g77` since it is available on many common platforms.

```
g77 -o first  first.f      or    f77 -o first first.f
         ↑       ↑
        (1)     (2)
```

(1) is the name of the output binary file (executable) i.e. the actual program that runs.
(2) is the name of the Fortran source code file i.e. the code that is compiled to create (1).

To run the program type: `first`
and follow the prompts.

Example 1A (Fortran 90)
This is a simple example of a Fortran 90 program that you can type in. Save it as: example.f90
Unlike Fortran 77 you don't need to start in column 7.

```
! EXAMPLE 1
!
real,dimension(10)::a=(/(i,i=1,10)/)
real,dimension(10)::b,c
integer i
b=a +1.          ! Vector addition
c=a*sin(b)       ! Vector operation
print *, c       ! Same as write(*,*)c

do i=1,10
  write(*,*)i,a(i),b(i),c(i)
enddo
end
```

To compile the source code on `abyss`:

```
f95 -o example example.f90
```

On a Solaris machine e.g. atlas, use `f90`.

Note: There are no Fortran 90 compilers on the other Linux machines e.g. cove, vislab01, at this time.

Example 2
Compile `mean.f` i.e. `g77 -o mean mean.f`
Run the program by typing `mean` - use x.dat as an input file. Repeat for `mean2.f` that computes the mean using a subroutine (`getmean`) and a function (`getmean2`).

Example 3
Compile `second.f` – use x.dat as an input file.

Example 4

Compile `third.f` – use xy.dat as an input file. This writes out the input file in (a) free format (b) formatted (inline) (c) formatted with a `format` statement.

Example 5

Compile `arrayb.f` in the usual way:

```
g77 -o arrayb arrayb.f.
```

Now compile the program with array bounds checking:

```
g77 -ffortran-bounds-check -o arrayb arrayb.f
```

Note: Using the SUN `f77` compiler on atlas:

```
f77 -C -o arrayb arrayb.f.
```

With this checking the program will abort when it tries to process `x(6)` since `x` is defined for subscripts 1-5 only.

Example 6

Compile `unirand.f`. This demonstrates some common intrinsic (built-in) functions and subroutines i.e. `iargc(), getarg, getpid` and `rand`.

Example 7

Compile readcon.f .

Important: On a 64-bit machine like abyss you need to specify the additional compiler option `-m32` i.e. `g77 -m32 -o readcon readcon.f`

If your program involves binary input and output e.g. CMP files, then use this option.

This reads in a CMP (conmap) file (binary i.e. not text) and writes out the contents to a text file in two ways – compact and verbose. Also the program makes use of iargc() and getarg to allow the input and output file names to be given on the command line in a typical UNIX-like fashion. If you type: readcon you will get a usage message:

```
Usage: readcon conmapfile textfile
Example: readcon pmsl.ncep.96010112 cmp.lis
```

Note that the example may not correspond to your data. In your case use:

```
readcon pmsl.ncep.96060106 cmp.lis
```

and look at the contents of cmp.lis. You can also compare `readcon` with my program `readcmp`:

```
readcmp -F '(F10.4)' pmsl.ncep.96060106
```

To produce a global map of the CMP file (pressure data):

```
conmap7 -G pmsl.ncep.96060106  (creates a NCAR Graphics file called gmeta)
ctrans gmeta                   (displays gmeta on the screen; click on black screen)
```

See: `man conmap` – try the options `–N` or `–S` for northern and southern hemispheres.

A hardcopy can be created with:

```
g2ps gmeta                     (creates the Postscript file g.ps)
```

You can view the Postscript file g.ps with Ghostview – handy to check g.ps before printing it:

```
gv g.ps
```

To print the Postscript file:

```
lpr -Ppclab g.ps
```

Example 8

Compile `writescat.f`. Use x.dat as an input file and choose your own output file e.g.scat.dat. To produce a graph of the data in the output file and view it on the screen:

```
scatter -ds scat.dat
ctrans gmeta
```

Type: `man scatter` to get help on the NCAR Graphics program `scatter`.
You can print the text file in a convenient way with `a2ps`:

```
      a2ps -Ppclab scat.dat
```

Without the `-P` option the printout goes to the UNIX Lab printer (`-Punix`).

Example 9
Compile `statcon.f` . Type: `statcon` to get the usage:

```
      Usage: statcon inconfiles stat outfile
              stat= 1 (average) 2 (standard deviation)
      Example: statcon *.pmsl 1 pmsl.ave
```

There is a set of CMP (conmap) files in this directory of 6-hourly mean sea level pressure for a few days in June 1996. In UNIX notation they are: pmsl.ncep.9606*. To form the average (mean) pressure map we specify the list of pressure CMP files, the value of stat (give 1) and the output file name (outfile e.g. ave.cmp).Try:

```
      statcon pmsl.ncep.9606* 1 ave.cmp
```

Have a look at ave.cmp with `readcon` or `mm` (see example 7) or try:

```
      readcmp  -F '(F10.4)' ave.cmp | more   (listing to screen)
```

and plot it:

```
      conmap7 -S ave.cmp           (southern hemisphere display)
      ctrans gmeta
```

Example 10
These are a couple of examples on *common blocks*. Compile `cblk1.f` and `cblk2.f` . These programs demonstrate the use of the `common` and `equivalence` statements that are often used in complex software like the Melbourne University General Circulation Model (MUGCM) and NCAR Graphics programs like `conmap7`. You probably won't use these concepts in your own programs but you may encounter them in code written by others.

Example 11
The correlation examples are essentially variations on a theme to show different ways of accomplishing the same thing. All of the programs calculate the correlation *r* of two series (*x* and *y*) and its statistical significance *p*. The data file corr.dat contains two columns (*x* and *y*) where *x* is the annual extratropical cyclone count for 30-50 ºS for 1958-1997 and *y* is the southern hemisphere annual temperature anomaly for the same period. We make use of the function `probst` written by someone in the 'wider' world to compute the *p* value to show that you don't need to write all of your own code. This is one of the advantages of programming! The function `ilen` is one I wrote to find the last non-blank character of a character variable.

(a) `corr1.f`
Compile in the usual way. Most of the computations are in the main program `corr1` and with calls to functions `probst` and `ilen`. To run:

```
      corr1 corr.dat
```

(b) `corr2.f`
Compile in the usual way. This is more modular. Subroutine `readdata` reads in the input file and subroutine `corr` performs the correlation and calls `probst` and `ilen`. To run:

```
      corr2 corr.dat
```

(c) `corr3.f`

Compile in the usual way. This is comparable to `corr2.f` but employs a common block called `blk1` to transfer x and y between the main program and the subroutines. Another common block (`blk2`) transfers the input file name. Note the use of the include statement: `include 'corr3.h'` which ‘pastes’ the code from `corr3.h` into the source code during compilation. This is handy if you need to modify the common blocks since you only change `corr3.h`. To run:

```
      corr3 corr.dat
```

(d) `corr4.f`

This is a rearrangement of `corr2.f`. All of the source code files are in the subdirectory corr4.dir. The main program is stored as `corr4.f` and the subroutines and functions are stored as separate `.f` files. There are two ways we could compile this program:

(1) Firstly, we can use the `f77` or `g77` command directly:

```
      g77 -o corr4 corr4.f corr.f ilen.f probst.f readdata.f
               ↑       ↑         ↑
              (1)     (2)       (3)
```

(1) is the name of the output binary file (executable)
(2) is the source code file of the main program (the one with `program`)
(3) is the list of subroutines and functions in any order
To run:

```
      corr4 ../corr.dat
```

where the file corr.dat is in the directory above.

(2) Alternatively, we can use the `make` command. See: `man make` for help.
This involves the use of a special file called a ‘makefile’ which is normally called Makefile. There are many ‘rules’ implied in the use of `make` and a lot of variation in the structure. We have specified array bounds checking to be turned on with FFLAGS= `-ffortran-bounds-check`. After the above step has completed `corr4` will be present in this subdirectory. As part of the process there are object files (`*.o`). Type: `make clean` which goes to the section named clean and removes the object files. Type: `make test` to go to the section named test and print a message. Normally we don’t delete the object files. If we modify any of the source files and type: `make corr4` then only the ones we changed get recompiled and linked to form `corr4` – a big saving in time if you have many source code files.
Type: `make corr4` where corr4 is the name of the section in the ‘makefile’ where compilation takes place. This is equivalent to: `make -f Makefile corr4`
The file called Makefile is given below:

```
# Makefile for corr4
# Kevin Keay  3/2/99

# *** NOTE ***
# MUST use a tab for commands e.g. all:[tab]$(OBJS) - SPACES won't
work

MAKEFLAGS=

FC = g77
FFLAGS= -ffortran-bounds-check
```

```
OBJS=  corr.o  ilen.o  probst.o  readdata.o

all: corr4

corr4:     corr4.o $(OBJS)
      $(FC) -o corr4 corr4.o $(OBJS)

clean:
      rm -f *.o

test:
      echo Just a test!
```

A version that is suitable for the SUN f77 compiler is given in Makefile.sun.
The differences between the g77 and SUN f77 versions are:

```
 FC= g77                             (FC= g77)
 FFLAGS= -ffortran-bounds-check      (equivalent to SUN f77 -C)
 $(FC) -o corr4 corr4.o $(OBJS)      (FC is f77 here)
```

To compile: `make -f Makefile.sun`
As before to run the program: `corr4 ../corr.dat`