Handout Six

October 24, 2013

1 Fortran 90 Modules

So far you have only been introduced to one type of 'program unit', that is, the main program unit. At first, in the main program unit, you were shown how to write simple executable code and then later how to use a '`CONTAINS`' statement to include your own functions and subroutines. There is another form of program unit in Fortran 90 and it is called a '`MODULE`'. A module has a range of applications in Fortran 90 and it is good practice to use them. A module can be seen as a related collection of entities that can be made available to one or more other program units. In this case an entity could be data, functions or subroutines. So, for example, you could have a module that '`CONTAINS`' all the 'functions' and 'subroutines' you have written that relate to matrices. This matrix module could then be included in other program units.

1.1 The structure of a module

Like a program unit a module has a 'typical' structure. This structure is similar to the that of main program unit.

```
MODULE <module name>

  <USE [other modules]>

  IMPLICIT NONE

  <Specification Section>

  CONTAINS

  <module procedure one>
  <module procedure two>
     :           :
  <module procedure  n >

END MODULE <module name>
```

- A module has its own name like any other program unit.
- A module can also make use of any entities in other modules, so you could have a '`USE`' statement between the module header and the '`IMPLICIT NONE`' statement.
- A module has its own '`IMPLICIT NONE`' statement.
- In the specification part of a module you can declare any data that you want to be 'globally' visible to all the procedures defined later in the module.
- A module can contain procedures (functions and subroutines) by placing them between a '`CONTAINS`' statement and the '`END MODULE`' statement.

- Modules are compiled separately from the main program unit so you should put them in a separate file. You can group more than one module in a single file but in most cases it is sensible to use a separate file for each module.
- Procedures that are contained in a module automatically have access to any other procedures contained in that module.

1.2 Making 'USE' of a module

In order to gain access to 'SUBROUTINES', 'FUNCTIONS' and data that exist in a given module Fortran provides the 'USE' statement. The 'USE' statement can be included in any program unit or procedure. The syntax can be structured in **three** forms.

1.2.1 The general 'USE' statement

USE <module_name>

This makes accessible all the available entities of the module 'module_name' to the program unit. All accessible procedures and data types declared in the module would be made available. See the example below where the module 'moduletest' hots only a function 'fact' and a subroutine 'bisect'.

```
PROGRAM modtest

  USE moduletest !***  Include "fact" and "bisect"

  IMPLICIT NONE

  REAL :: fact_local
  LOGICAL :: bisect_local=.TRUE.
  INTEGER :: a=5
  REAL :: b=0,c=1

  fact_local=fact(a)
  IF (bisect_local) CALL bisect(b,c)

END PROGRAM modtest
```

Note that the 'USE' statement is included before the 'IMPLICIT NONE' statement.

Exercise One: In a directory 'handout6/exercise1' Copy the three files 'example.f90', 'example_mod.f90' and 'Makefile' into your 'exercise1' directory using the command below (note the period at the end).

```
cp ~/info/examples/handout6/exercise1/* .
```

The main program file 'example.f90' makes 'USE' of the module file 'example_mod.f90'. Therefore the main program has access to the two functions 'sumsq' and 'reverse' that the module 'CONTAINS'. The two functions 'sumsq' and 'reverse' are not finished and so you need to finish writing the source code. The comments in the code explain the purpose of the functions. To compile the code, type 'make' at the fortran command line prompt and press the return key. The 'make' command here does the compilation on **both** 'f90' files and leaves the executable file 'example' in the directory. There is more information on 'make' later in the these notes.

1.2.2 The '`USE`' Statement with the rename '`=>`' option

It is possible to rename a data type or procedure in a '`USE`' statement. The reason being that in some cases the '`USE`' statement, as above, could cause problems. If a local data item in the program unit using the module is declared with the same name as an entity declared in the module then this would cause a name conflict. A conflict could also exist if a program unit uses more than one module and there is a name conflict between those modules.
For example if we had a module that '`CONTAINS`' a procedure called 'fact' and a program unit that uses the module has a local data type declared with the name '`fact`', then '`USE`' of this module as in the example above will result in a conflict! To get round this problem Fortran 90 uses a re-name symbol '`=>`' as in;

`USE` <module_name>, <local_name> => <name_in_mod>[, <local_name2 > => <name_in_mod2>]

The small example program below uses the module '`moduletest`' and renames the function '`fact`' in the module to be called '`factorial`' locally. It also renames the subroutine '`bisect`' in the module to be called '`bisection`' locally. It does this as both '`fact`' and '`bisect`' already exist as local variables in the main program unit and hence avoids any name conflicts!

```
PROGRAM modtest

  USE moduletest, factorial => fact, bisection => bisect

  IMPLICIT NONE

  REAL :: fact
  LOGICAL :: bisect=.TRUE.
  INTEGER :: a=5
  REAL :: b=0,c=1

  fact=factorial(a)
  IF (bisect) CALL bisection(b,c)

END PROGRAM modtest
```

1.2.3 The '`USE, ONLY :`' statement

It is clear that it would be advantageous if there was a way to 'select' which entities we would like to '`USE`' from a module. Not surprisingly there is an easy way of doing this and it involves including the '`ONLY`' attribute inside the '`USE`' statement.

`USE` <module_name>, `ONLY :` <name1>[,<name2>,.....]

So for the above example we could have.

`USE moduletest, ONLY : fact, bisect`

Note that the renaming '`=>`' can also be used in the comma separated list, for example

`USE moduletest, ONLY : factorial => fact, bisection => bisect`

This means we make available from the module '`moduletest`' only the entities `fact` and `bisect` and, in the process, rename them locally `factorial` and `bisection`.

1.3 Modules used to hold common data

Modules can be used to pass data between program entities. This is done by declaring the commonly used data in the specification section of the module. It is then simply included in the desired procedure or program unit using the '`USE`' statement.

```
MODULE commondata

  IMPLICIT NONE

  SAVE
  REAL :: pi=3.14159
  INTEGER :: married=10, age=45

END MODULE commondata

! *************************************************************

PROGRAM example

  USE commondata, ONLY : age, married, pi

 IMPLICIT NONE

 PRINT*,age,married,pi

END PROGRAM example
```

This method of common data transfer between different sections of code should be not be liberally used instead of argument lists. However, it is often the case that many argument lists in the same code can get very long and complicated and full of the same common data elements. It is in cases like this that common data can be passed to the different sections through modules in this way.
Note that following the '`IMPLICIT NONE`' statement there is a '`SAVE`' statement. This is recommended in Fortran 90 to ensure that the data values are preserved between various program units making use of the module. You should always include it in a module when you declare data.

1.4 Some reasons for using Fortran 90 modules

We have just looked at how we can use modules to provide useful entities such as functions, subroutines and also common data types to other program units. Aside from this modules have other uses.

- Modules help keep the main program file at a tidy manageable size.
- Modules allow for better 'data protection' by limiting the scope of main program variables.

1.5 An Example Code

The first section of the code is the 'module'. This module section goes in a file of its own and should be named appropriately, '`sin_module.f90`' or '`sin_mod.f90`' for example.

```
MODULE series_routines  !***** example module *****

  IMPLICIT NONE

CONTAINS

  FUNCTION expand_sine(x) !*** Finds series expansion

    REAL, DIMENSION(:) :: x
    REAL, DIMENSION(SIZE(x)) :: expand_sine

    expand_sine=x-x**3/factorial(3)+x**5/factorial(5)-&
                x**7/factorial(7)+x**9/factorial(9)

  END FUNCTION expand_sine

! *****************************************************

  FUNCTION factorial(n)
  !*** calculates factorials

    INTEGER, INTENT(IN) :: n
    REAL :: factorial, a
    INTEGER :: i

    a=1.0
    DO i = 1,n
       a = a*i
    END DO

    factorial=a

  END FUNCTION factorial

END MODULE series_routines
```

The next section of code is the main program unit and this also is best placed in a file of its own. An appropriate name could be something like '`sin_main.f90`'. Of course we still have to compile our source code and then link the object code into an executable file. Note that in earlier codes all the source code was written into a single file and the compiler created an executable file in a single pass. With more than one file, the compiler will need more passes to create the executable. Firstly for each source code file we will need to compile and create an object file. Then all the object files will be linked together to create the executable file. To make this process easier we will be making use of '**makefiles**'. A '**makefile**' is a set of instructions to the Linux '**make**' command. The purpose of the 'make' utility is to determine automatically which pieces of a large program need to be compiled or recompiled, and issue the correct commands to do so. The usage of the '**make**' command will be explained in more detail shortly.

```
PROGRAM sin_main

  USE series_routines, ONLY : expand_sine

  IMPLICIT NONE

  REAL, PARAMETER :: pi = 3.14159265359
  REAL, DIMENSION(20) :: x, series
  INTEGER :: i

  DO i=1,20   !*** Define the range of x values to use
     x(i)=2.0*pi*(i-1)/ 19
  END DO

  series=expand_sine(x) !****  sin(x) up to five terms

  !**** Prints the results to the screen
  PRINT'(2e12.4)', (x(i), series(i), i = 1, 20)

END PROGRAM sine_main
```

Some points to note about the code.

- In the main program after the call to '`expand_sine`' we use a '`PRINT`' statement to write the results to the screen. This statement contains an example of an implied '`DO`' loop. They fit on a single line and are convenient in input/output (io) statements and also for initialising arrays. For example the two following statements produce identical arrays for '`vec2`' and '`vec3`'.

  '`INTEGER ::  i`'

  '`INTEGER, DIMENSION(5) ::  vec2=(/ (i,i=1,5) /)`'

  '`INTEGER, DIMENSION(5) ::  vec3=(/ 1,2,3,4,5 /)`'

  This statements declare the arrays and also initialise the elements to be '`1,2,3,4,5`'.

  The syntax of an implied DO is the following:

  '`( item-1, item-2, ...., item-n, DO-var = initial, final, step )`'

  '`( item-1, item-2, ...., item-n, DO-var = initial, final )`'

  It starts with a '`(,`' followed by a set of items, separated by commas, followed by a '`DO`' variable, an equal sign, an initial value, a final value, and a step-size, and ends with a '`)`'. Like a typical '`DO`' loop, if the step size is 1, it can be eliminated.

  Depending on the place where an implied '`DO`' is used, the items can be variables, including array elements, or expressions.

  The meaning of an implied '`DO`' is simple: For each possible value of the '`DO`' variable, all items (i.e., item-1, item-2, ..., item-n) are listed once and adjacent items are separated by commas. The following small code will help you follow how the syntax of the implied '`DO`' loop work.

```
PROGRAM implied
!**
!** Example of an implied DO loop

  IMPLICIT NONE

  INTEGER :: i
  REAL, DIMENSION(8) :: aa=(/(i,2*i,i=1,4)/)

  PRINT'(f4.1)',(aa(i),i=1,8)

END PROGRAM implied

!* The output to the screen is as below

 1.0
 2.0
 2.0
 4.0
 3.0
 6.0
 4.0
 8.0
```

The above code '`sine_expansion`' demonstrates how you can use modules to build a code in a modular fashion. You can copy this code, and any files needed to compile it, to a directory in your home area by typing

```
cp ~/info/examples/sinmod/* .
```

in the directory you want to copy the files to. Do not forget the dot ('.') at the end to indicate you want the file to be copied to the directory you are currently in. To compile and run the code you will have to read the next section that introduces the basic use of a '`Makefile`'.

1.5.1 Compiling more than one file : '`Makefile`'

If you copied the files as directed in the above section you will now have in your directory a file called '`Makefile`'. This file allows you to compile the two source code files and link together the object files (created by the compilation) into a single executable by just using a single command '`make`'. Try it, just type '`make`' and hit [`Return`]. You can look at the file '`Makefile`' in '`nano`'. You do not need to understand any of the main part you just have to know how to change the first section that contains the lines 'main=sin_main' and 'mod1=sin_module'.

```
#########################################################
# REPLACE the main_prg_name below with the name of
# your main program without the ".f90" extension
# REPLACE the module_name below with your module
# name without the ".f90" extension

# PROGRAM NAME (no .f90)
main=sin_main
# MODULE NAME  (no .f90)
mod1=sin_module

# compiler options
opt1=-check all -check noarg_temp_created
opt2=-check nouninit  -traceback
#########################################################
cmplr=f90 $(opt1) $(opt2)
objects1 = $(mod1).o $(main).o

$(main)   : $(objects1)
	$(cmplr) -o $(main) $(objects1)
$(mod1).o    : $(mod1).f90
	$(cmplr) -c $(mod1).f90
$(main).o  : $(main).f90 $(mod1).f90
	$(cmplr) -c $(main).f90

clean :
	rm -f *.mod *.pcl *.pc *.o $(main) *.inc *.vo *.d
```

The 'Makefile' contains 'required' tabs at the beginning of certain lines so please copy 'cp' this file around instead of typing it in yourself!

This makefile will compile and link any code that has a main program file and a module. At the moment the 'Makefile' is setup for a main program file called 'sin_main.f90' and a module file called 'sin_module.f90'. You will want to compile and link codes that have different names for the main program unit and the 'Makefile'. To do this you must copy the 'Makefile' to the directory containing the main program file and the module it uses. Then change the lines 'main=sin_main' and 'mod1=sin_module' to 'main=<newname1>' and 'mod1=<newname2>'. Where 'newname1' and 'newname2' are your main program file name and your module file name without the '.f90' extension.

Class Project :: Part Three :

Create a new directory 'class_project/part3' for your matrix library code and copy your class project ('part2') fortran file into it. Rearrange your matrix fortran source code file into two 'new' files. One main program file and the other a 'MODULE' file containing all the matrix related procedures you have written in 'part2'. Call the module 'part3_mod.f90'. Create a main program file called 'part3.f90' that does the same work as your class project 'part2'. Copy the 'Makefile' from the 'sinmod' program into this directory and edit it in 'nano' so you can compile and run 'part3'. As before you just type 'make' to compile your code.

```
cp ~/info/examples/sinmod/Makefile  .
```