FORTRAN 95 Review

prepared by Eric Salas

!

These are the very basics, there is so much more to FORTRAN that you can use and should look up to improve your coding skill

Compiling Your Program

You'll want to use a good text editor to write your code in. I use Programmer's Notepad, but most any text editor will work.

To Compile, you will open the command prompt, and change directories to where your files are saved

From the terminal,

>>gfortran yourfilename.f90 –o –name.exe

Or

>>g95 yourfilename.f90 –o –name.exe

Flags

Basic Structure

Declaration

PROGRAM

IMPLICIT NONE

Execution…

Termination

END PROGRAM

```
PROGRAM machine_epsilon
IMPLICIT NONE
REAL :: eps
INTEGER :: k
eps = 1.0
DO k = 1, 1000
IF(eps+1.0<=1.0) EXIT
eps = eps/2.0
END DO
eps = 2.0*eps
WRITE(*,*) 'Machine Epsilon=', esp
END PROGRAM
```

Variables and Constants

REAL

INTEGER

CHARACTER

LOGICAL

```
REAL :: x
REAL :: y1, y2
INTEGER :: i, j, k
CHARACTER :: file_name

REAL, PARAMETER :: n = 10.0
```

Parameter sets a variable that will stay constant throughout the program

Operations

Add	+
Subtract	–
Multiply	*
Divide	/
Exponent	**

```
DO i = -10, 30, 1
   x = REAL(i) / 10.
   y = (x**2) - (3.*x) + 2.
END DO
```

! Always Watch for Mixed-Mode Arithmetic

Basic I/O

READ(*,*)

WRITE(*,*)

(location, format)

```
WRITE(*,*) 'enter value of x'
READ(*,*) x_in

X_out = x_in**2

WRITE(*,*) 'x squared =', x_out
```

DO Loops

```
DO
    …
END DO

name: DO
    …
END DO name

DO i = 1, 10
    …
END DO
```

```
outside: DO j = 1, max_iter
   calculate: DO i = 1, n
   Tnew(i) = ((1./2.)*( Told(i+1) + Told(i-1) )) + &
   & ((1./(8.*k(i)))*(Told(i+1)-Told(i-1))*(k(i+1)-k(i-1)) )+ &
   & ( (q(i) *(dx**2)) / (2.*k(i)) )
   END DO calculate
   s = 0.
   error: DO i = 0, n + 1
      s = s + (  (Tnew(i)  - Told(i))**2 )
   END DO error
   Told = Tnew
END DO outside
```

IF/THEN

IF (Logical) THEN

statement

END IF

```
IF (ierror /= 0) THEN
   WRITE(*,*) 'Could not open file'
   STOP
END IF
```

IF (Logical) *statement*

```
IF (sqrt(s) < err) EXIT
```

Writing to Files

```
OPEN(unit=10,file=filename,status='old',action='read',iostat=i_err)
```

OPEN(

Unit = The file number. Always pick a number greater than 10, this is what the code will reference for the file

File = this is the actual name of the file you are either creating or opening

Status = 'old' if the file already exists, 'new' if you're a creating a new one, or 'replace' if you want to do just that, replace the existing file.

Action = what will you be doing with the file ? 'read' or 'write'

iostat = assigns a value to given variable(i_err in this case). If the file was successfully opened, iostat will return a zero. Remember to declare i_err!

Don't forget to close your file when you are done!

Writing to Files

Example

```
OPEN(unit=10, file='output.dat', action='write', status='replace')

WRITE(10,'(I2, 8F10.5)') i, xl, f(xl), xu, f(xu), xr, f(xr)

CLOSE(10)
```

Don’t forget to close your file when you are done!

Arrays

Arrays are groups of variables or constants

Declared by adding the DIMENSION attribute to a variable type

TYPE, DIMENSION(n)

N can be a value (n) or a range (a:b)

For multiple dimensions, add a comma (m,n)

```
REAL, DIMENSION(10) :: x
INTEGER, DIMENSION(-5:5) :: array
REAL, DIMENSION(10,10) :: matrix1
```

Arrays

An array is a collection of values in memory. An array, A, of dimension(10) will set aside ten slots:

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

While an array that has dimension (3,3) will look like a matrix:

A(1,1)	A(1,2)	A(1,3)
A(2,1)	A(2,2)	A(2,3)
A(3,1)	A(3,2)	A(3,3)

Filling an Array

You can assign a whole array to one value

```
INTEGER, DIMENSION(3) :: x
x = 0
```

0	0	0

Or assign individual locations

```
x(2) = 7
```

0	7	0

Or use an array construct

```
x = (/12, 2, 6/)
```

12	2	6

Allocatable Arrays

*When you want to change the size of an array as part of the program, add the property **allocatable** to the declaration.*

Use a colon in place of a number for the size of every dimension (:) or (:,:)

*Then **allocate** the size later*

```
REAL, ALLOCATABLE, DIMENSION(:) :: A
REAL, ALLOCATABLE, DIMENSION(:,:) :: B

ALLOCATE(A(-1:1))
ALLOCATE(B(100,100))
```

And it is good practice to deallocate your array when you are done, and can be required for some algorithms

```
DEALLOCATE(A)
DEALLOCATE(B)
```

Procedures

Procedures include Subroutines and Functions. Procedures are included in the code after all of the main declaration and execution with the statement **CONTAINS.**

A function takes in specified values and returns only one value.

A subroutine may return several or no values.

```
PROGRAM
IMPLICIT NONE

declaration
execution

CONTAINS

FUNCTION …
END FUNCTION

SUBROUTINE …
END SUBROUTINE

END PROGRAM
```

Subroutines

Subroutines must be declared by

SUBROUTINE name(input values)

And they have their own declaration section for the variable that will be passed to the subroutine as well as any intermediate variables the subroutine will use.

After the declaration section is the execution section just like normal.

Then finish with

END SUBROUTINE

A subroutine is accessed within your code by stating:

CALL subroutine_name(input variables)

Subroutine Example

```
CONTAINS
                                        ! for writing matrices
   SUBROUTINE writematrix(arr, m, n)
   INTEGER, INTENT(in) :: m,n
   REAL(wp), INTENT(in), DIMENSION(m,n) :: arr

   DO i = 1, m
      WRITE(*,*) '|', (arr(i,j), j = 1, n), '|'
   END DO

END SUBROUTINE
```

Functions

Funcitons work like subroutines, but return only one value.

FUNCTION name(input_values)

The function itself must be declared as a real, integer, etc

The final statement must assign what the output value for the function will be

function_name = some_value

Make sure you avoid unintended side effects!

Finish with

END FUNCTION

Function Example

```
CONTAINS

FUNCTION gravity(m1, m2, r)
   REAL :: gravity
   REAL, INTENT(in) :: m1, m2, r
   REAL :: G

   G = 6.672_wp * (10.0_wp**(-11))
   gravity = (G*(m1)*(m2)) / (r**2)

END FUNCTION
```

Modules

Modules are useful for organizing your code and for storing values and procedures. Think of a module as a box that you can store things in which can be accessed by different codes. For example, values of constants you use all the time (like π or e), or common subroutines, like a root finding method.

A module looks a lot like a program:

MODULE mod_name

IMPLICIT NONE

Declaration

CONTAINS

subroutines

END MODULE

Modules

A program accesses a module with the USE statement, which comes before the IMPLICIT NONE statement

PROGRAM whatever_you_named_it

USE name_of_mod

IMPLICIT NONE

And when you compile, you must include the name of your module in the command.

>gfortran system.f90 program.f90 –o –run.exe

Module Example

```
MODULE system_mod
IMPLICIT NONE

INTEGER, PARAMETER :: sp = selected_real_kind(p=6)
INTEGER, PARAMETER :: dp = selected_real_kind(p=15)
INTEGER, PARAMETER :: ep = selected_real_kind(p=18)
INTEGER, PARAMETER :: qp = selected_real_kind(p=30)

INTEGER, PARAMETER :: wp = dp

END MODULE
```

Double (or more) Precision

Double precision increases the size (uses more memory) of a variable, making operations more accurate

Single Precision	**Double Precision**
π = 3.1415927	π = 3.141592653589793

There are 3 ways to use double precision:

1. **Compile** all floating point (real) variables with double precision:

   >>gfortran code.f95 –o code.exe –fdefault-real-8

   >>g95 code.f95 –o code.exe –r8

2. Assign a Working Precision(selected_real_kind)

   REAL(wp) :: mypi = 3.141592653589793 (where wp represents your working precision)

   3.141592693589793_wp (see module on previous slide and other examples in this presentation)

3. Use the DOUBLE PRECISION declaration instead of REAL

   REAL :: mypi = 3.14159

   DOUBLE PRECISION :: myotherpi = 3.141592653589793