Quick Guide to Fortran

A.J. De-Gol
University of York
Autumn Term 2007

This document is intended as revision of the first year Fortran course, it covers the most basic program operations needed to be able to write simple programs. It is no way intended as a complete description of the language; resources for this will be provided at the end. It must be stressed that there is always more than one way to solve a problem with programming; developing a style that you are comfortable with can be as important as good coding practices.

NB: In Fortran there is no difference between CAPITALS and lowercase; the compiler will read them equivalently. Some consider only using lowercase to be more '1337', others find it hard to hit the shift key on their laptops.

1 Basic Program

All programs need a beginning and an end. Remember to always include IMPLICIT NONE, as this ensures strong typing, and will help to make sure that your program does what you think it is doing. IMPLICIT NONE must also included in subroutines, functions and modules.

PROGRAM programe
    IMPLICIT NONE

    ! NB: this line is not 'seen' by the compiler
    ! Anything following ! is commented out.
    ! Use comments _often_ and your marks and understanding will increase.

END PROGRAM programe

This is your most basic, complete program; it accomplishes nothing but to exist, however, all puzzles must be solved starting with the first piece, so if you can’t think of where to begin, then make sure you have at least this. Remember that when executing your program the computer looks at what is in between PROGRAM and END PROGRAM and executes as it goes down, in that order. You may only have one “PROGRAM”, though you may have as many subprograms (subroutines, functions) as you like
2 Binary Operations

To do calculations in a program you will need to know what syntax corresponds to an equivalent mathematical operator:

+ : add e.g. a+b
- : subtract e.g. a-b
* : multiply e.g. a*b
/ : divide e.g. a/b
**: raise to a power e.g. a**b (a.k.a. exponentiation)

Ordering is also very important; this is like when entering long sums into a calculator. In order of what will be evaluated first we have: Powers, multiplication, division, additions and subtraction. For example:

4.0**2*3.0/4.0+1.0 = 16.0*3.0/4.0+1.0 = 48.0/4.0+1.0 = 12.0+1.0 = 13.0

To avoid confusion one may use brackets, in which case anything within the brackets will be evaluated before anything else.

3 Variables

3.1 Types and Declarations

Manipulating variables is the bread and butter of programming (understanding variables requires knowledge of algebra, \( a = b + c \), for example). The types of variables we will be concerned with over this course may be split into three branches: integer, real and boolean/logical. NB: Fortran can also handle complex numbers (kind=16) as well as derived types (see resources).

Declaring of all variables is done before any operations (e.g. \( a = b \) is an assignment operation and \( 1.0*2.0 \) is a multiplication operation), and after IMPLICIT NONE:

\[
\text{real :: a, vari, a_var} \\
\text{integer :: b} \\
\text{logical :: c}
\]

Note the syntax. TYPE followed by :: and then the variable name/s. To declare more than one variable of the same type, simply separate the variable names with a comma. Variable names must start with a letter, after which they can contain numbers and a few select special characters, e.g. underscores.

Integers and reals are handled differently by the machine. Integers take the form \(-1,0,1,2,3\) etc, they are purely whole numbers. Their main use comes in as: loop counters; declaring array sizes; assigning unit numbers to external files; indices algebra etc.

Real numbers are of the form: \(1.89769876, 7896.3423400, 1.12e10\) and so on, anything with a decimal place; they are predominantly used for calculations. The last example \((1.12 \times 10)\) is scientific notation... Fortran is predominantly used for high-performance science simulations!

Boolean or logical variables work as polar opposites: true/false; 1 or 0, on/off, etc. They are used to test if a condition is true or false and then to act upon the result.
To assign a value to a variable, after it has been declared, is done by stating
the variable you wish to change, then an ‘=’ sign and then the assignment, for
instance:

```plaintext
a = 1.903
var1 = 14. ! note the single ‘.’ to indicate as var1 is real
b = 2 ! no dot as b is integer
a_var = 2.0**2 ! This is 2 squared: note the decimal point is absent on the power term.
c = .true. ! logical declaration, alt: .false.
```

3.2 Parameters

When you have a number that doesn’t vary, e.g. \( \pi \), you may declare it as
a parameter. It will be unchangeable throughout the programs course. The
declaration is the same as any variable but with an addition after the type.

```plaintext
real, parameter :: pi = 3.14159265
integer, parameter :: MaxNumOfPlayers = 100
```

3.3 Precision

So far we have been using what is known as single precision variables; these
have well defined limits dependent on machine architecture and OS. But what
happens when you want to represent (as we will) much bigger numbers, with
much more precision? The answer is to use double precision variables. My
prefered way of using double precisions is as follows: Declare a double precision
parameter (dp - one point zero ‘d’ zero) and then use this in all other declarations
by putting the variable name in brackets after variable type:

```plaintext
integer, parameter :: dp=kind(1.0d0) ! Do this before ANY others
real(dp) :: a, b ! a and b are now double precision
```

NB: There is a rounding error pitfall to watch out for here. If variable ‘a’
is double precision, one must be careful when using it to make use of it being
double. That is:

```plaintext
a = 0.0
```
May give a to be 0.0000000248192746. The extra precision contains whatever
was previously in memory. Therefore we can make sure it is double precision
zero (0.0000000000000000) by appending a dp tag to the end:

```plaintext
a = 0.0_dp
```
Note the underscore and then the dp. For clarity, you may call your DP variable
anything, for example:

```plaintext
integer, parameter :: example = kind(1.0d0)
real(example) :: aVariable
aVariable = 0.0_example
```
Is equivalent to before.

See: SELECTED_REAL_KIND and SELECTED_INT_KIND for even more precisional control.
3.4 Conversion

To convert between real and integer you may use a number of operations.

**REAL to INTEGER:**

```fortran
integer, parameter :: dp=kind(1.0d0)
real(dp) :: a
integer :: b

b = NINT(a) ! Nearest Integer
b = INT(a) ! Standard conversion; rounds DOWN.
b = CEILING(a) ! if a=1.9; b=2
b = FLOOR(a) ! if a=1.9; b=1; same as INT
```

**INTEGER to REAL**

```fortran
real :: a
integer :: b

integer, parameter :: dp=kind(1.0d0)

a = REAL(b) ! If b = 2; a = 2.00000000
a = REAL(b,dp) ! for double precision, use this!
```

**NB:** Be careful with integer and real division! Often by default, mixed real and integer operations will be converted to purely real by the compiler, but this cannot be assumed. Always convert to real and never do integer/integer division. Why?

In summary to this section: **ALWAYS USE DOUBLE PRECISION REALS and ALWAYS CONVERT TO REAL when doing mixed, integer/real operations.**

4 Subroutines

Subroutines and functions are known collectively as 'subprograms' but behave in slightly different ways. Subroutines are perfect for separating a program into its component parts. By analogy, if I were to build a [simple] car I would need: a body; wheels; steering/control and an engine/power. Therefore I would break my program 'car' up into those self-contained, sub-parts; this makes it easier to take on each component individually.

Subroutines are extremely versatile and can generally do anything a program can do (including calling subroutines), but you can have more than one. Subroutines can have information passed to and from them, in what are known as dummy arguments. Here is a simple subroutine called by the main program, all it does is take two numbers, add them, and give the main program back the answer.

```fortran
PROGRAM progone
```
IMPLICIT NONE
! declare variables
integer, parameter :: dp=kind(1.0d0)
real(dp) :: a,b,c,d,e,f
a = 1.0
b = 2.0
d = 1.0
e = 3.0

! Call subroutine and pass a,b,c
call addsubber(a,b,c)
! c now equals 3.0
! Note I can reuse the same subroutine with different numbers
call addsubber(d,e,f)
! f now equals 4.0
END PROGRAM

! After program has ended I can declare my subroutine.
SUBROUTINE addsubber(x,y,z)
IMPLICIT NONE
! Local variable declarations
integer, parameter :: dp=kind(1.0d0)
! Declare Dummy variables
real(dp), INTENT(IN) :: x,y
real(dp), INTENT(OUT) :: z

z=x+y
END SUBROUTINE

Some very important things must be noted here:

- **CALL** is used when accessing subroutines.

- **ARGUMENTS** passed into the subroutine are in brackets after the name of the subroutine called, they must be of the same type and in the same order as where you declare the subroutine.

- **INTENTS** are essential. They decide how a variable passed into the subroutine is to be used. **You only need to declare intents INSIDE subroutines.** - Not programs OR functions! They can be of 3 types: **IN, OUT or INOUT**. Use IN if the variables are used but don’t change inside the subroutine; use OUT for anything you purely wish to RETURN and use INOUT if you are passing something into a subroutine AND it will change whilst there.

- **DUMMY VARIABLES**. You may be asking how in the program I declared a,b and c; yet in the subroutine there was only x,y,z. This is allowed because the subroutine is CLOSED and doesn’t ‘know’ about the a,b and c of the main program. It only knows there will be **3 numbers** all reals, it will take the first two number, add them, assign this as the third and then return the third number. This is why they are known as dummy
variables; I could have called them a, b and c in BOTH the subroutine and the program.

- **REUSING** the subroutine with different numbers is allowed as with d, e and f above.

A few further things about subroutines: you can pass arrays into them, they don’t even need to have arguments if you use modules (be careful with global declarations!)

### 5 Functions

Functions, as mentioned, are also subprograms. They differ from subroutines in how they are used and they need to be declared anywhere you wish to use them. It is best to consider them mathematically: for instance if I wanted a function $f(x) = \cos(x) + \sin(x)$. I will give the function a name (myFunc), I want to assign the variables $y=f(x=2)$ and $z=f(x=3)$ in a program. I do so as follows.

```fortran
PROGRAM myProg
  IMPLICIT NONE
  ! Function Declarations:
  real, external :: myFunc ! Real as myFunc returns REAL number.
  ! Variable declarations:
  real :: y,z, j
  ! Use function:
  y = myFunc(2.)
  ! Alternatively:
  j = 3.0
  z = myFunc(j)
END PROGRAM

REAL FUNCTION myFunc(x)
  IMPLICIT NONE
  real :: x
  ! Note the VALUE of the FUNCTION uses the FUNCTION NAME
  myFunc = COS(x) + SIN(x)
END FUNCTION
```

This bears repeating: **Note the VALUE of the FUNCTION uses the FUNCTION NAME.** Inside the myFunc function I give what the function is equal to by using the function name: myFunc = $f(x)$. Inside the main program I use the function as though IT IS A NUMBER, making sure to declare it at the top. You can pass more than one argument into a function.
5.1 Intrinsic Functions

There are many functions which are included in Fortran (compiler dependent for extras) these include, in a non-exhaustive manner:

\[
\begin{align*}
\text{SIN}(x) & \quad \text{Returns sine of } x \\
\text{COS}(x) & \quad \text{Returns cosine of } x \\
\text{EXP}(x) & \quad \text{Returns } e^x \\
\text{SQRT}(x) & \quad \text{Returns } \sqrt{x} \\
\text{ABS}(x) & \quad \text{Returns absolute value of } x \text{ e.g. } \text{abs}(-29)=29
\end{align*}
\]

You can see they work much as their mathematical equivalents; most trigonometric functions (\text{ATAN} = \text{arctan}) are included. See reference books for further information.

6 Modules

Modules become increasingly important with larger programs or bigger software (i.e. group) projects. You may declare variables, arrays, functions, subroutines etc... in a module and then \textit{any program or subprogram you wish to 'share' the variables just has to 'use' the module containing them.}

For small level programs, modules are useful if you are certain that there will be no multiple instances of the same variable declared, and don’t wish to keep re-declaring the same variables. \textbf{Modules must be declared BEFORE program}. A often reuse modules for common constants:

\begin{verbatim}
MODULE uni
  IMPLICIT NONE
  SAVE
  ! Universal constants, eg pi, double precision=dp etc...
  ! Other than dp and sp, all parameters have uni_ as prefix to signify universal constant
  integer, parameter :: dp=kind(1.0d0), sp=kind(1.0)
  REAL(DP), PARAMETER :: uni_PI=3.141592653589793238462643383279502884197_dp
  REAL(DP), PARAMETER :: uni_TWOPi=6.283185307179586476925286766559005768394_dp
  real(dp), parameter :: uni_DegreeConverter=uni_PI/180.0_dp
  real(dp) :: uni_1overroot2pi
  real(dp) :: uni_emass=9.1093826*10e-31, uni_echarge=1.60217653*10e-19
END MODULE

MODULE progmod
  use uni
  IMPLICIT NONE
  SAVE

  real(dp) :: CommonVar
END MODULE

PROGRAM myProg
\end{verbatim}
use uni
use progmod
IMPLICIT NONE

! This is a local declaration
real(dp) :: a

a = 0.5*uni_emass

call mySubRout

! CommonVar now equals 2.0
END PROGRAM

SUBROUTINE mySubRout
use uni
use progmod
IMPLICIT NONE

CommonVar = 2.0_dp

END SUBROUTINE

Note the only declarations outside of modules are for LOCAL variables only, for instance, above, the subroutine mySubRout does not 'see' or 'know about' the variable 'a', however, after the subroutine is called, BOTH the program and SUBROUTINE (and anywhere else that uses 'progmod') see the 'CommonVar' has changed to 2.0.
To summarise:
• MODULES must be written before anything else.
• You utilise a module with the use command followed by the module name.
• 'USE modname' is done BEFORE IMPLICIT NONE
• Modules may 'use' other modules. NB: in above, as 'progmod' uses 'uni' anything using 'progmod' will also be using 'uni'.
• See reference material for the more information on modules.

7 Arrays
An array can be seen as an ordered set of values stored contiguously (i.e. all connected in memory); if it helps you can think of arrays as vectors or matrices, or pictorially as a co-ordinate system on a map:
If I take a 2x2 box. I can refer to to any element of the box using an (x,y) cartesian coordinate system; the first element corresponding to (x=1,y=1) and the last to (x,y)=(2,2)
Using the diagram above, if I asked you which element of the diagram below corresponds to the smaller black circle, you would reply with...........? And the small white circle...........?

In this second diagram we have assigned VALUES to array elements. Okay, so it is only in pictorial form, however, we could still refer to the VALUES the array element contained using the (x,y) system. So if I said box(1,2) it would be the same as saying 'nested circles'. This is how we use arrays in fortran, only the box can be one dimensional or more.

PROGRAM myProg
IMPLICIT NONE

! Variable Declarations
integer, parameter :: dp = kind(1.0d0)

! Array Declarations:
integer, dimension(2,2) :: intray
real(dp), dimension(-10:10) :: realray

! Advantages of arrays are you can do WHOLE ARRAY OPERATIONS:
! Without specifying a 'coordinate', EVERY element will be addressed
intray = 0 ! This sets ALL elements to 0
realray = 0.0_dp ! As does this but 0.00000000...

END PROGRAM

Here I have declared a two-dimensional INTEGER ARRAY (therefore it can contain values which are integers: -2874, 34, 1, 2 etc) and a one-dimensional, double precision, REAL ARRAY. Lets now look at the declarations themselves. We can see that the declaration is very similar to variable declarations with the added DIMENSION(from:to,from:to) statement, where 'from' and 'to' are integers. By default Fortran starts arrays from 1: so that 'intray', from above, has dimension(1:2,1:2), but as we can see, it is possible to declare arrays to start from virtually anywhere. The dimensions themselves are separated by a comma; a 3-D array would have DIMENSION(x,y,z).
To access an array element we use its 'coordinates' as was discussed above:

\[ \text{intray}(1,1) = 1 \]
\[ \text{intray}(1,2) = 2 \]
\[ \text{realray}(-4) = 292387.23232323 \]
\[ \text{realray}(0) = \text{real}(	ext{intray}(1,1), \text{dp}) \times 16.0_{\text{dp}} + \text{realray}(7) \]

7.1 Allocatable Arrays

There are different types of arrays, even boolean, but what about if I don’t yet know how big I want my array? I know I need to have an array but am not sure how big it is. In these cases it is possible to use ALLOCATABLE ARRAYS. There are a couple of fundamental differences in both the behaviour and declarations of these arrays.

\[ \text{real} \left( \text{dp} \right), \text{allocatable}, \text{dimension} (:, :) :: \text{myArray} \]

The declarations require the additional ‘allocatable’ statement, but when it comes to dimension we need to use colons and commas to indicate how many dimensions it is to have, that is its shape. As you can see from the previous example, a 2-D array has been declared. At some later time, I find out that I wish to have the dimensions of length \( n \) and \( m \) respectively, where \( n \) and \( m \) are integers we have discovered (or a user inputted). Now I use the allocate() statement:

\[ \text{allocate} \left( \text{myArray}(n,m) \right) \]

I am now free to use myArray like any other array; care must be taken to deallocate the array once its use has completed, to prevent memory leaks. I can de/allocate more than one array at a time, separating using commas.

\[ \text{deallocate} \left( \text{myArray} \right) \]

Allocatable arrays have advantages in both the size of array allowed can be bigger, and in some instances they are quicker. But by far the greatest strength lies in their flexibility.

7.2 Intrinsic Array Operations

There are many intrinsic functions to act on arrays, a select few which may be of use are:

- \text{SUM} \quad \text{! Sets 'a' to be the sum of all element values in array}
- \text{MAXVAL} \quad \text{! Returns the minimum value in an array}
- \text{MINVAL} \quad \text{! Returns the maximum value in an array}
- \text{MAXLOC} \quad \text{! Returns location(element number) of maximum value}
- \text{MINLOC} \quad \text{! Returns location of minimum value}
- \text{ALLOCATED} \quad \text{! Returns allocation status of an array}
- \text{TRANSPOSE} \quad \text{! Transpose an array onto another array with switched dimensions}
  \quad \text{! NB: array must be of rank 2}
- \text{CSHIFT} \quad \text{! Circular shift all elements (to left or right)}
- \text{EOSHIFT} \quad \text{! End of shift all elements (but loses one end and other end is 0.0)}

There are many more intrinsic functions to arrays (See pg. 464 of Lahey) and each function above is able to take additional arguments.
8 IF and SELECT CASE statements

IF statements (and later SELECT CASE) provide the basis for logical program control. An IF statement must always have a corresponding END IF. They work as follows:

IF (statement to be true) THEN
   !ACTIONS IF STATEMENT TRUE
ELSE IF (other condition) THEN
   !ACTIONS IF FURTHER CONDITION MET BUT FIRST CONDITION NOT
ELSE
   !ACTIONS IF STATEMENTS ARE FALSE
END IF

or

IF (statement to be true) THEN
   !ACTIONS IF STATEMENT TRUE
ELSE
   !ACTIONS IF STATEMENTS ARE FALSE
END IF

or

IF (statement to be true) THEN
   !ACTIONS IF STATEMENT TRUE
END IF

There are a couple of ways to give the '(statement to be true)' bit: if using BOOLEAN/LOGICALS we can just use the variable name.

```logical :: testvar = .true.
if(testvar) then
   print *, 'It IS true'
else
   print *, 'It’s the LIES I can’t stand!'
end if```

In this example the program will print, to the screen: “It IS true” as we set testvar as true; continue to note there is no difference between capitals and lowercase. Conversely, one can use the .NOT. statement to mean ‘execute if this condition is false’. Ordinarily however we may want to say: do this if x is greater than y.

8.1 Greater/Less than, Equal to

IF(x.gt.y) THEN
   ! operations to do
END IF

Forgetting the THEN statement is a constant source of error. Check your syntax! There are many ways of giving the logical arguments, including symbolically. It is of personal preference to use letters as I find them easier to remember, a few examples:
For example, if we wish to say ‘If x is in the range (0-4)’ we could use:

\[
\text{IF}(x \geq 0 \text{ AND } x \leq 4) \text{ THEN }
\]

Please see as many sources of literature as you can until you understand logical constructs. There are many additional conditions and methods that have been purposefully omitted for brevity.

### 8.2 SELECT CASE

A SELECT CASE statement (or simply CASE statement) may be crudely regarded as a glorified, nested, IF statement. Its benefits arise when you have a multi-conditional operation. If opens with `SELECT CASE(test variable)` and ends with `END SELECT`. Say if we have an integer, called ‘test’: if its value is 1 or 2 then set ‘x’ to be equal to ‘test’, else if `test = 3` set ‘x’ to equal 25, else if `test = 4` set ‘x’ to be 49; for all other values of ‘test’ set \( x = 0 \).

```
SELECT CASE(test)
    CASE DEFAULT
      x=0
    CASE(1:2) ! This means 1 to 2.
      x=test
    CASE(3)
      x=25
    CASE(4)
      x=49
END SELECT
```

We can immediately see this becomes much clearer than using IF statements. CASE statements aren’t limited to INTEGER values, we can even use CHARACTERS (for more information, nothing beats a good book). This next example insults people based on what day of the week it is; note how the DEFAULT value will be a week day.

```
CHARACTER(LEN=256) :: day
SELECT CASE(day)
    CASE DEFAULT
      print *, "Enjoy work, loser!"
    CASE("Wednesday")
      print *, "Did you get the memo requiring all TPS reports to have covers?"
    CASE("Saturday")
END SELECT
```
print *, "Enjoy the hangover!"
CASE("Sunday")
    print *, "Sunday, bloody Sunday... how's that report for Monday coming?"
END SELECT

9 DO loops

DO loops may be seen as the 'engine' of a program, they are used for repeating operations, incrementing operations, acting on consecutive elements of an array etc... You begin with DO counter=start,end,steps where steps is optional and conclude with an END DO.

NB: Always make sure to use INTEGERS with loop counters.

program dotest
    implicit none
    integer :: i, a,b,c
    
    do i=1,10,2
        print *, i
    end do
    
    a=1
    b=10
    c=2
    do i=a,b,c
        print *, i
    end do

end program

The output from above would be to print “1,3,5,7,9” to the screen, twice. As stated, the 'steps' statement is optional, it is somewhat traditional to use ‘i’, ‘j’ and ‘k’ (in that order) for DO loops.
DO loops may be nested inside each other, the inner loops will complete first.

```fortran
  do i = 1, imax
    do j=1,jmax
      do k=1,kmax
        print *, i,j,k
      end do
    end do
  end do
```

If 

$$\text{imax} = \text{jmax} = \text{kmax} = 2$$

the output of the previous program would be:

- i,j,k
  - 1,1,1
  - 1,1,2
  - 1,2,1
  - 1,2,2
  - 2,1,1
  - 2,1,2
  - 2,2,1
  - 2,2,2

### 9.1 Infinite DO loops

Sometimes you just want a program to run 'until it’s done'. You don’t know how many timesteps are needed, or how many points you wish to go over, it may be an unspecified domain: the point is the program must run until it has the answer, then stop. Fortunately there are special cases of DO loops that *never stop until YOU tell it to* (always remember the ultimate 'off switch' is to pull the power from the wall! - also NEVER do this!), these are **Infinite DO loops**. The syntax is like that of an ordinary DO loop but with: no counter, no start point, no stop point and no steps:

```fortran
  DO
      x = x + 1 ! take value of x, add one, assign that number to x
  END DO
```

This program will never end, it will keep adding 1 to x; even when x is as big as it can be – in which case it will 'cycle' over and flip to the *lowest* it can be – and so on. Infinity is a long time, luckily computers are patient enough to wait for it; unluckily they are patient enough to wait for it when we are not.

So, how do we make these things stop? We do so by way of an **EXIT** command: when a condition has been met (i.e. when the answer has been found or some preset limit has been reached), remember we use **IF** statements to check this, then you may simply type **EXIT**. Be careful as anything else inside the loop you wanted to do before exiting must be written before the EXIT command.
DO
    if(condition met) then
        ! Anything to do before leaving loop
        ! do here.
        EXIT
    end if
END DO

Be aware that EXIT will only quit the loop it is enclosed by.

DO
    DO
        if(condition met) then
            exit
        end if
    END DO
END DO

Note that this is still an infinite DO loop since we only exited the innermost loop. Be careful!

In Linux, if you suspect your program to be trapped in an infinite loop, type 
CTRL-C to quit the program - do this from the command line where the program was executed.

10 Input

When you want the USER to input the value of a variable during the running of the program, use the READ statement.

program readtest
    implicit none

    integer :: b

    print *, "Please enter a value for b:"
    read *, b

    print *, b
end program

The program accepts a value 'b' from the user and prints it back to confirm, try it for yourself. The READ statement is very useful as it means that you don't have to re-compile to change one variable, it is also extremely powerful when combined with allocatable arrays; for instance, declare an allocatable array, allocate it based on the users input and then fill every element in that array as the REAL value of its INDEX (that is it's corresponding, 1-D 'coordinate').
program readtest
  implicit none

  integer :: b, i, imax
  real, allocatable, dimension(:) :: oneDarray

  print *, "Please enter a value for b:"
  read *, b

  allocate(oneDarray(1:b))

  do i = 1, b
    oneDarray(i) = real(i)
  end do

  deallocate(oneDarray)
end program

11 Output

We have seen, without explanation, copious use of the PRINT statement. This merely prints whatever follows it to the screen; for multiple things, separate with a comma. Make sure you include the *, after PRINT, this ensures standard formatting. Enclose strings(words) in quotation marks or apostrophes. Examples:

  print *, "hello, the variable x is", x, 'which is nice'
  print *, x, y, z

11.1 External Files

Perhaps you wish some data to be more permanent than just being written to the screen; you may even wish to draw a graph in your life; computers are very good at them. In order to WRITE to an external file you need a UNIT number:

  write(22,*) x, y, z

  This will write x, y and z (separated by a standard delimiter such as space or tab) into the file FORT.22 (filename is compiler dependent). The UNIT in this case was 22; certain units are reserved, as a rule of thumb: start above 20. If you wish to do something more exotic/professional then you require an OPEN statement with corresponding CLOSE statement when you are finished using it.

program writetest
  implicit none

  integer :: i
  integer :: ios ! this is a standard name for error checking
OPEN(UNIT=22, FILE="myFile.dat", iostat=ios)
IF(ios.ne.0) STOP "Error in opening file"

do i = 1,10
   WRITE(22,*) real(i), real(i)**2
end do

CLOSE(UNIT=22,iostat=ios)
IF(ios.ne.0) STOP "Error in closing file"

end program

This program will print x Vs. x*x (this is x*x, or ‘x squared’) to the file ‘myFile.dat’. The iostat=ios part may be excluded and the program will still run, but you should always include it (as well as the IF statement) to check that your program is working as you expect it to. As with all functions, such as WRITE, there are many other options which may be included to give more control to the programmer; I once more state that these are available in many good books (some of which are free online).

12 Resources

Books:

• Ellis, Lahey and Phillips, Fortran90 Programming.
  This is the recommended book. For others, see below.

Internet:

• http://www-users.york.ac.uk/ hcb1/fortran.html
  Paddy Barr’s online course - you should be familiar with this reference. It is
  an extensive guide with many programming exercises.

• http://en.wikipedia.org/wiki/Fortran
  This is a good article; however it will also point you in the direction of
  many free books and online guides that can help you. Not to mention
  COMPILERS, GRAPHICS LIBRARIES and even CODE REPOSITORIES. See near the bottom of the page.

Do remember that there is never one Holy-Grail of resources, find a
book that works for you and increase your understanding, the syn-
tax will naturally follow. Most of all BONNE CHANCE!