

# Introduction to Fortran 90

Dragos B. Chirila (cdragos@awi.de)

December 7, 2009

## Contents

|   |           |
|---|-----------|
| <b>1 Fortran 90</b>   | <b>2</b>  |
| 1.1 History . . . . .                                       | 2         |
| 1.2 About This Document . . . . .                           | 2         |
| <b>2 Fortran Basics</b>                                     | <b>2</b>  |
| 2.1 Basic Program Layout . . . . .                          | 2         |
| 2.2 Source Format . . . . .                                 | 3         |
| 2.3 Constants & Variables . . . . .                         | 4         |
| 2.4 Basic I/O . . . . .                                     | 6         |
| 2.5 Program Flow Elements (IF, CASE, Loops, etc.) . . . . . | 8         |
| 2.6 Arrays . . . . .  | 10        |
| 2.6.1 Writing Compact Code . . . . .                        | 11        |
| 2.7 Subprograms . . . . .                                   | 11        |
| 2.7.1 Functions . . . . .                                   | 12        |
| 2.7.2 Subroutines . . . . .                                 | 12        |
| 2.8 Intrinsic Functions & Subroutines . . . . .             | 14        |
| <b>3 Working with multiple files (optional)</b>             | <b>15</b> |
| <b>4 Appendix B: Tables</b>                                 | <b>19</b> |

# 1 Fortran 90

## 1.1 History

The FORTRAN (contraction from “mathematical FORMula TRANslation system”) language was created in the '50s by John Backus of IBM labs. It was the first successful HLL (High-Level Language) and was quickly adopted for numerical computing. Over time, many other HLLs appeared (Cobol, Pascal, C & C++); nonetheless, the FORTRAN compilers consistently outperformed those of other languages in numerical computing for over 20 years. The new languages have pros and cons; as far as computational scientists are concerned, they all suffer from the fact that they are not targeted towards numeric computing, while Fortran is<sup>1</sup>. However, they do offer some useful features. Hence, Fortran suffered a series of revisions in order to keep up with the current trends and to benefit from the recent computing architectures. The most significant recent revision is Fortran 90<sup>2</sup>. Fortran 90 is backwards compatible with the 77 version, which makes it easy to use the vast amount of legacy code available.

Fortran has a quite long history, and this sustained the common misconception that the language itself is “ancient”. This is not necessarily true; unlike previous versions, Fortran 90 (the one we are interested in) adopted a free source format and added many features one expects for a modern programming language.

## 1.2 About This Document

This document is meant to be a short “getting started” guide for Fortran 90, and is targeted mainly towards beginners in numerical computation. Of course, this guide is by no means comprehensive, but it should provide, at least, the necessary starting steps needed for the course assignments. Still, for more in-depth information, the reader can consult any one of the excellent books available on the topic (see [Hahn, 1996] or [Michael Metcalf, 1999]).

The topics discussed herein are followed by real programs, which one can compile and test with.

# 2 Fortran Basics

## 2.1 Basic Program Layout

The basic layout of a Fortran 90 program is shown below:

```
PROGRAM [program name]
  IMPLICIT NONE
  [variables declarations]
  .....
  [main program executable code]
  .....
  CONTAINS
  [functions & subroutines code]
  .....
END PROGRAM [program name]
```

**Discussion of the code:** The words in capital letters are Fortran keywords. Throughout this text, we will always use capital letters for the language keywords, but this is not required. The schematized program listing is self-explanatory, except for the

---

**IMPLICIT NONE**

---

<sup>1</sup>Of course, it is not our purpose here to discredit those programming languages. What we are discussing is choosing the right tool for the task at hand. Fortran is quite inappropriate for writing operating systems, but it excels at numerics.

<sup>2</sup>There were further revisions (Fortran 95 & Fortran 2000), but they didn't introduce significant novelties.

line, which tells the compiler not to try an automatic type assignment of the variables. For now, all you need to keep in mind is that it is good programming practice to include this statement, as it avoids many hard to spot bugs in complex programs.

## 2.2 Source Format

The most basic units of each Fortran program are the 26 letters of the English alphabet, the 10 Arabic numerals (from 0 to 9), the underscore (`_`) and the so-called *special characters*. Fortran 90 is a case-insensitive language, i.e. no distinction is made between the lower and upper-case letters (thus, if you declare the variables *field* and *fiEld* in the same program unit, the compiler will complain about the variable being defined twice). The special characters in the standard are given in Table 2 (*Appendix B*).

These special characters can be combined to form special characters sequences (such as operators or separators), examples of which will appear as we go along with our tutorial.

**Comments:** Commenting your programs is highly recommended, and can be done in Fortran 90 by typing an exclamation mark (!). This will cause the preprocessor to treat the rest of the characters after ‘!’ on that line as a comment, i.e. the line will be ignored by the compiler.

**Multi-line instructions:** Each line of code in Fortran 90 may contain up to 132 characters. If an instruction does not fit into these limits, it can be continued on the following lines (up to 39 continuation lines can be used) by inserting an “&” (ampersand), as in the code listing below: <sup>3</sup>

```
PROGRAM CONTINUATION_1
  IMPLICIT NONE
  INTEGER :: number
  number = 312 &
           +3  ! The result is 315
  PRINT*, number
END PROGRAM CONTINUATION_1
```

**Gap-free multi-line:** The syntax of the continuation markup described above inserts a space after the last character on the line to be continued. This is not what you want if your last characters on the line to be continued form a part of a constant value or name or a Fortran keyword<sup>4</sup>; in this case, you will have to add another ampersand on the following line where the remainder of that token is, such as in this code listing:

```
PROGRAM CONTINUATION_2
  IMPLICIT NONE
  INTEGER :: number
  number = 31&
           &2+3  ! The result is still 315. Of course, this is ugly
                ! code and should be avoided.
  PRINT*, number
END PROGRAM CONTINUATION_2
```

**Indenting:** Spaces can be freely used to indent your code, as long as they do not split program tokens. Hence,

<sup>3</sup>Don't panic if there are unfamiliar things in the code listing. We will come to them soon.

<sup>4</sup>These will be further referred to as "program tokens" for conciseness.

```
a=b+10
```

and

```
a      =      b      +      10
```

are entirely equivalent from the compiler's point of view.

**Multi-lines combined:** Each program line generally contains one instruction. However, it is possible to combine many short lines separated by a semicolon (;) as one single line:

```
a=b; b=c; c=a ! Note that a semicolon is not
              ! required after the last instruction
```

## 2.3 Constants & Variables

A programming language wouldn't be of much use unless it allows you to define and manipulate the entities of your problem (such as velocity, temperature, etc.). This brings us to the topic of *data types*. A data type is comprised of a set of data values (e.g. integer numbers), means of denoting those values (e.g. 0, 100, -50) and a set of operations defined on the set (e.g addition, multiplication, etc.). The Fortran 90 standard requires every compliant compiler to provide 5 built-in data types<sup>5</sup>: 3 numeric (INTEGER, REAL and COMPLEX) and 2 non-numeric (CHARACTER and LOGICAL). Each data type has an associated *KIND*, which is basically a way to specify the precision to be used (examples for this will follow).

We will focus on the INTEGER, REAL and CHARACTER data types, as these are the ones you will use most often.

- *INTEGER* : Values of this type can be any integer number (such as 0, 123, -321), within some bounds depending on the system<sup>6</sup> we are working with and on the *KIND* of integer we request. Variables of this type have to be declared as shown below:

```
INTEGER    i      ! Simple Variable Declaration
INTEGER :: i=10   ! Declaration and Initialization.
              ! Note that the double colon '::' IS REQUIRED
              ! in this case (not required for simple declaration)
```

The second line shows how to contract the declaration and the initialization on a single line. You will generally need to initialize the variables you will use, because the declaration (which only reserves memory for the variable somewhere) leaves them in an unpredictable state, depending on what residual data was present at that address.

**The operations:** allowed for the INTEGERS are the usual arithmetic ones: *addition* (+), *subtraction* (-), *multiplication* (\*), *division* (/) and *raising to a power* (\*\*).

- *REAL* : Examples of valid values of this type are 3.14, .14, 31., 1.E2<sup>7</sup>, or 3.14D4<sup>8</sup>; again, the higher and upper bounds of the usable numbers are dependent on the system and *KIND*<sup>9</sup>. Variables of this type are declared as follows:

<sup>5</sup>You can also define your own data types, with their own set of operators, very much like in C++.

<sup>6</sup>For example, on a 32-bit computer, the range could be from  $-2^{31}$  to  $+2^{31} - 1$

<sup>7</sup>If you have not seen this type of notation before, it is called the "exponential notation of real numbers". In our case, 1.E2 means  $1.0 \times 10^2$ .

<sup>8</sup>The "D" has the same role as the "E" above, but it marks a different precision (double)

<sup>9</sup>On a typical implementation, REALs range from  $-10^{38}$  to  $10^{+38}$ .

```

REAL    x          ! Simple Variable Declaration
REAL :: x=1.23    ! Declaration and Initialization
REAL :: x=1.E2   !

```

**The operations:** allowed for the REALs are the same as those mentioned above for the INTEGERS.

- *CHARACTER*: Constants of this type are the alphanumeric characters. The variables are initialized in a similar manner:

```

CHARACTER    my_character    ! Simple Declaration
CHARACTER :: my_character='c' ! Declaration + Initialization

```

Attention: normally, you would not be interested in storing single characters, but character strings. To accomplish this, you would need to store more than one character (i.e. a *string*). It can be done quite easily by specifying a length parameter (*LEN*):

```

PROGRAM read_write_line
  IMPLICIT NONE
  CHARACTER(LEN=80) line ! LEN specifies the length of our string.
                          ! In this case, we set it to 80, which
                          ! should print nicely in most Unix terminals.

  PRINT*, 'Enter some text (a word or a string enclosed in &
  .....a pair of apostrophes or quotes)'
  READ*, line ! Input should be a single word or a string within "" or ''
               ! e.g. Encyclopedia OR "The weather is just fine..."
               ! If more than one word is supplied, e.g
               ! The sun is rising
               ! reading will stop at the first space encountered, i.e.
               ! we will have line="The".

  PRINT*, line
END PROGRAM read_write_line

```

**KIND:** Let us now develop a little on the *KIND* selection mentioned above. You can think of the *KIND* parameter as a way of adjusting the precision of the intrinsic data types. The standard only requires that at least one *KIND* is available for each intrinsic data type. Hence, the existence of other *KIND*s except for the default one depends entirely on your compiler implementation. We specify the *KIND* of a variable while declaring it:

```

INTEGER (KIND=2) I

```

To see what integer kinds are available for your compiler, you can run the following program:

```

PROGRAM detect_kinds
  ! Detects Integer KINDs available
  IMPLICIT NONE
  INTEGER k,n
  n=0
  DO
    n=n+1
    k=SELECTED_INT_KIND(n)

```

```

    if(k .EQ. -1) EXIT
    PRINT*, n, k
    ! Where k represents the KIND needed to represent
    ! an integer number in the range  $-10^n..10^n$ 
  END DO
END PROGRAM detect_int_kinds

```

```

! To analyze available KINDs for REAL, replace
k=SELECTED_INT_KIND(n)
! with
k=SELECTED_REAL_KIND(n)
! in the program.

```

## 2.4 Basic I/O

In most of the examples so far, we had to do some Input/Output (I/O) operations to communicate with our programs. The most simple form of I/O in Fortran is the so-called *list-directed I/O*. This corresponds to using the *READ\** and *PRINT\** statements. By default, these instructions read (write) from (to) the terminal, but as we will soon see they are also useful for *File I/O*. Their basic usage is shown below:

- *Output with PRINT\** :

```

SYNTAX: PRINT*, [list of things to print]
SAMPLE PROGRAM:

PROGRAM simple_print
  IMPLICIT NONE
  PRINT*, 'Hello_world!'
END PROGRAM simple_print

```

- *Input with READ\** :

```

SYNTAX: READ*, [list of things to read]
SAMPLE PROGRAM:

PROGRAM your_name
  IMPLICIT NONE
  CHARACTER (LEN=20) name
  PRINT*, 'Enter_your_name_(inside_apostrophes_or_quotes,_of_course):'
  READ*, name
  PRINT*, 'Hello_', name, '!'
END PROGRAM your_name

```

A word of caution on the usage of the *READ\** instruction: each *READ\** instruction expects a new line. If you pressed *Enter* and it didn't read all it was supposed to read, it will scan for the rest of the input on the following line. But you cannot enter on a line the input for several adjacent *READ\**. Thus, while it is perfectly fine to write something like:

```

PROGRAM readsome
  IMPLICIT NONE
  INTEGER a, b, c
  READ*, a, b, c

```

```

PRINT*, 'a=', a, '          b=', b, '          c=', c
END PROGRAM readsome

```

with the Input: 2 3 4 [Enter].

On the other hand, the program:

```

PROGRAM readmore
  IMPLICIT NONE
  INTEGER a, b, c
  READ*, a
  READ*, b
  READ*, c
  PRINT*, 'a=', a, '          b=', b, '          c=', c
END PROGRAM readmore

```

will only work as expected with an input formatted as:

2[Enter]

3[Enter]

4[Enter]

as you can test by yourself.

The same *READ/PRINT* instructions can be used for *File I/O*. For this, one needs to initialize the file first. This is done using the *OPEN* instruction, which assigns a number (called *file descriptor*) to the actual file on the disk. Then, we need to use that file descriptor as a parameter for our I/O instructions, as in the following example:

```

PROGRAM file_io
  ! Reads three numbers from the file 'data.in' and prints them
  ! in reverse order in the file 'data.out'.
  ! Of course, you will have to create the file 'data.in'
  ! (and write 3 integers in there)
  ! prior to running this program...
  IMPLICIT NONE
  INTEGER a, b, c
  OPEN(UNIT=10, FILE='data.in') ! links 'data.in' on the disk with file
descriptor 10
  OPEN(UNIT=20, FILE='data.out')
  READ(10,*) a, b, c
  WRITE(*,*) 'a=', a, '          b=', b, '          c=', c
  WRITE(20,*) c, b, a
  CLOSE(10); CLOSE(20)
END PROGRAM file_io

```

We generally recommend to use values greater than 10 for the file descriptors, since several values below this range are linked by default to the standard input/output.

**Note:** In actual scientific programs, you will have to match the format of your output datafiles to the requirements of the postprocessing tools you want to use (plotting utilities, other programs, etc.). In the listing above, the program wrote the three numbers on a single line of the file 'data.out'. If, for example, you want each number on its own line, you could use three *PRINT* commands:

```

PRINT(2,*) c
PRINT(2,*) b

```

```
PRINT(2,*) a
```

However, this is not the most effective approach. It is worthwhile to invest some time to learn about *Formatted I/O* from one of the references.

## 2.5 Program Flow Elements (IF, CASE, Loops, etc.)

There are several ways to control the flow of the programs in Fortran 90. In real-life situations, we often have to distinguish between different cases, and execute the appropriate sequence of operations for each case. In all of these cases, we need a criterion for the different possible situations, and this brings us to the topic of *branching*. Usually, decisions are taken based on the value of a logical tests (e.g. “is  $x_i \leq 5$ ?”), which reduce to either TRUE or FALSE.

**IF:** The most basic test is the familiar *IF* construct, with the syntax:

```
GENERAL SYNTAX:
IF ([logical_condition_1]) THEN ! THEN is optional when there is
                                ! no ELSE branch
    [instructions_1]             ! Instructions to execute if           ! Required
                                ! [logical_condition_1] is TRUE
ELSE                             ! OTHERWISE                          ! Optional
    [instructions_2]             ! Execute these...                 ! Optional
END IF                           !                                  ! Required
```

### REAL-LIFE EXAMPLE

```
PROGRAM odd_or_even
  IMPLICIT NONE
  INTEGER number
  PRINT*, 'Enter an integer number: '
  READ*, number
  IF (MOD(number,2) .EQ. 0) THEN ! MOD is an intrinsic function which returns
                                ! the remainder of the division of 2
integers.
    PRINT*, number, ' is even.'
  ELSE
    PRINT*, number, ' is odd.'
  ENDIF
END PROGRAM odd_or_even
```

The *IF construct* is useful when we do not have too many cases to take care of. Note that more than one logical condition can be verified using so-called *nested IF* constructions, i.e. by inserting other IFs inside one of the branches (THEN or ELSE) of the top-level IF construct. This practice is allowed in Fortran 90, but it tends to lead to programs which are difficult to follow.

**CASE:** If we have many situations we have to test, it is more elegant to use the *CASE construct*, exemplified below:

```
GENERAL SYNTAX:
SELECT CASE (expression)
  CASE (case_1) ! If expression=case_1 ! At least one
                ! case should be tested
```

```

[instructions_1]      ! Execute these instructions      ! Optional
CASE (case2)        ! Otherwise, if expression=case_2 ! Optional
[instructions_2]      ! Execute these instructions      ! Optional
.....
CASE DEFAULT       ! If none of the cases above holds ! Optional
[default_instructions] ! Execute these default instructions ! Optional
END SELECT

```

**EXAMPLE PROGRAM** (a more elegant version of the previous program):

```

PROGRAM odd_or_even2
  IMPLICIT NONE
  INTEGER number
  PRINT*, 'Enter an integer number:'
  READ*, number
  SELECT CASE (MOD(number,2)) ! Calculates the rest of number divided by 2
                               ! (See the Subsection "Intrinsic Functions")
    CASE (0)
      PRINT*, number, 'is even.'
    CASE DEFAULT
      PRINT*, number, 'is odd.'
  END SELECT
END PROGRAM odd_or_even2

```

**Loops:** Sometimes, we may want our program to carry-out the same set of instructions for a number of times. This requires us to use a *loop*. There are two basic types of loops: *deterministic* (when we know exactly the number of repetitions) and *non-deterministic* (when we don't have this information). Both cases are programed in Fortran 90 using the *DO construct*:

- *Deterministic loops:*

```

GENERAL SYNTAX:
DO variable=expression_1, expression_2 [, expression_3]
  [block_of_instructions]
END DO

```

In the listing above, *expression\_1* and *expression\_2* represent the boundaries of the range from which *variable* will take values from, while *expression\_3* represents the increment. The increment is entirely optional (and defaults to 1 if it is omitted), yet it is useful in some cases. Let us suppose that our task is to calculate the sum of all *even* numbers from 0 to some integer *N* (given by the user). The following code would do the job:

```

PROGRAM even_sum
  IMPLICIT NONE
  INTEGER :: N, i, sum=0
  READ*, N
  DO i=0,N,2
    sum=sum+i
  END DO
  PRINT*, 'Sum is: ', sum
END PROGRAM even_sum

```

- *Non-deterministic loops*: We don't always know the exact number of steps we will have to execute. A classical example is the so-called *number guessing* game. You pick a number (we limit our sample code to "digit guessing"), and the computer tries to guess it by asking whether his guess is the correct one, smaller or larger than the number you picked. The game ends when the computer guessed your digit. Of course, we do not know here how many guesses the computer will need to finish the task. A program for this problem is the following:

```

PROGRAM digit_guess
  IMPLICIT NONE
  INTEGER    :: try=5
  REAL       R
  CHARACTER  :: feedback='u' ! Initialize with an unaccepted response.
  LOGICAL    :: guessed=.FALSE.
  PRINT*, 'Please pick a digit between 0 and 9'
  PRINT*, 'I will guess it ...'
  DO
    IF(guessed .EQV. .TRUE.) EXIT
    PRINT*, 'Is it', try, '? (y for Yes / n for No)'
    READ*, feedback
    SELECT CASE(feedback)
      CASE('y')
        guessed=.TRUE.
      CASE('n')
        PRINT*, 'Is your number larger or smaller? &
.....(l for larger / s for smaller)'
        READ*, feedback
        IF(feedback .EQ. 's') THEN
          try=try-1
        ELSE try=try+1
        ENDIF
      CASE DEFAULT
        PRINT*, 'Invalid response. Trying again ...'
    END SELECT
  END DO
  PRINT*, 'Done! Exiting ...'
END PROGRAM digit_guess

```

## 2.6 Arrays

You will often have to work with arrays. These are *compound objects* (i.e. they can hold more than one value) of the same type and KIND specifier. Arrays are usually derived from the intrinsic data types, but arrays of a derived data type are also possible (although not discussed here). Arrays are characterized by their rank (number of dimensions) and their size (number of elements). For example, a 1-dimensional array of 20 REAL elements could be declared as:

```
REAL, DIMENSION(20) :: X
```

Its elements can be referred to using the syntax  $X(i)$ , where  $i$  is an *INTEGER* in the range  $1 < i < 10$ .

**Note:** Unlike other programming languages (such as C or C++), the default value of the index corresponding to the first array element is **1** and **NOT 0**.

Fortran has another interesting feature: one can specify bounds to the running index. Hence, it is possible to declare an array as:

```
REAL, DIMENSION(-10:9) :: Y ! Y is now a 1D array with 20 elements,
                             ! the first one of which is Y(-10)
```

However, you will often need to use multi-dimensional arrays. Fortran supports up to 7 dimensions, and the syntax for n-dimensional arrays is quite similar to the example above:

```
! [data type], DIMENSION([n numbers separated by commas]) :: [name of array]
.....

! Example: 2-D array of 10*20 integer values:
INTEGER, DIMENSION(10,20) :: temperature

! The temperature at e.g. mesh point (5,7) can be addressed in the usual way:
PRINT*, temperature(5,7)
.....

! Example: 3-D array of 10*10*10 real values:
REAL, DIMENSION(10,10,10) :: speed

! Where the element (2,3,4) is addressed in the usual way:
PRINT*, speed(2,3,4)
```

### 2.6.1 Writing Compact Code

Fortran90 allows you to save keystrokes and space on the screen by several code contraction techniques. The one most often encountered is demonstrated in the example below:

```
DO i=istart ,iend
    field(i) = 20.
ENDDO
```

The main idea here is that we can avoid the DO-loop entirely. Hence, the code above becomes a one-liner:

```
field(istart:iend)=20.
```

This is especially helpful in larger applications, where the DO-loops would unnecessarily complicate the code. Furthermore, the contraction allows the compiler to apply some clever optimizations to our program.

## 2.7 Subprograms

Sometimes, the same set of instructions has to be carried-out many times during the execution of a program. In these situations, it makes sense to group those instructions into a *subprogram*. Fortran 90 supports two types of subprograms: *FUNCTIONs* and *SUBROUTINEs*. These can be *internal* (in the same file as the *main program* (and hence compiled along with it) or *external* (in separate files). The external subprograms can be grouped in *modules*, which can be used by more than one program (see Section “Working With Multiple Files”). For now, we shall restrict our discussion to *internal subprograms*.

Internal subprograms appear between the *CONTAINS* keyword (see first code listing) and the final line of the program:

```
END PROGRAM [program name].
```

Also, note that subprograms may not have the *CONTAINS* keyword, thus subprograms may not have subprograms of their own (although they can call other subprograms).

### 2.7.1 Functions

The syntax of a FUNCTION subprogram is given below:

```
FUNCTION Name ([argument list])
  [function's declaration statements]
.....[function's executable statements]
END FUNCTION [Name]
```

Functions take as input the *argument list* and return a single value (or an array). Thus, it is necessary to assign a type to the returned value, as it is done in the following code:

```
PROGRAM Factorial
  IMPLICIT NONE
  INTEGER N

  PRINT*, 'Enter the natural number whose&
.....factorial should be calculated:'
  READ*, N
  PRINT*, 'Result:', Fact(N)

CONTAINS

  FUNCTION Fact(X)
    INTEGER Fact, X, Tmp, I
    Tmp = 1
    DO I=2, X
      Tmp=I*Tmp
    END DO
    Fact=Tmp
  END FUNCTION Fact

END PROGRAM Factorial
```

Any variable declarations inside the function (except the *return value*) are irrelevant for the main program, as they cease to exist once the function returns control to main. The *return value* is fed back to the main program and can be assigned to a variable or manipulated as usual.

### 2.7.2 Subroutines

Although subroutines have uses similar to functions, there are also some different aspects:

- No return value is associated with the name of the subroutine, thus it need not be declared;
- Subroutines are invoked via a *CALL* statement;
- The keyword *SUBROUTINE* is used instead of *FUNCTION*;
- A subroutine need not have any parameters.

The syntax of a subroutine is:

```
SUBROUTINE Name ([argument list])
  [subroutine's declaration statements]
.....[subroutine's executable statements]
END SUBROUTINE [Name]
```

Generally, subroutines are useful when more than one variable is to be returned from the subprogram, as in the example below:

```
PROGRAM swap_numbers
IMPLICIT NONE
REAL :: a=2., b=3.

PRINT*, 'Before_subroutine_call:'
PRINT*, "a=", a
PRINT*, "b=", b
CALL swap(a,b)
PRINT*, 'After_subroutine_call:'
PRINT*, "a=", a
PRINT*, "b=", b

CONTAINS

SUBROUTINE swap(x,y)
  REAL, INTENT(INOUT) :: x, y
  REAL x, y, aux
  aux=x; x=y; y=aux
END SUBROUTINE swap

END PROGRAM swap_numbers
```

It is possible to enforce some rules on the information flow in/from a subroutine using the *INTENT*-specifier. *INTENT(IN)* tells the compiler that the variable is only read but not written, *INTENT(OUT)* says that the variable is only written to, while *INTENT(INOUT)* allows both operations. This is something we recommend, since it turns the compiler checks to your advantage, if you are attempting by mistake to modify the wrong variable from within a subroutine.

## 2.8 Intrinsic Functions & Subroutines

The Fortran 90 standard defines a set of *intrinsic functions* for some common operations (such as the function MOD in the listing *odd\_or\_even2* above). Calling an intrinsic function proceeds in the same way as for user-defined functions. A list with the most frequent intrinsic functions is given in Table 1 below:

Table 1: Intrinsic Functions & Subroutines in Fortran 90 (selection from [Michael Metcalf, 1999])

| Name                            | Description  |
|---------------------------------|--|
| $ABS(x)$                        | Absolute value.  |
| ACOS(x)                         | Arc cosine function.   |
| AINT(a [,kind])                 | Truncate to a whole number.  |
| ALLOCATED(array)                | True if the array is allocated.  |
| ASIN(x)                         | Arc sine function.   |
| ATAN(x)                         | Arc tangent function.  |
| CEILING(a [,kind])              | Least integer greater than or equal to its argument<br>( <i>kind</i> permitted only in Fortran 95) . |
| COS(x)                          | Cosine function.   |
| COSH(x)                         | Hyperbolic cosine function.  |
| DBLE(x)                         | Convert to double precision real.  |
| DOT_PRODUCT(vector_a, vector_b) | Dot product.   |
| DPROD(x,y)                      | Double precision real product of two default real scalars.   |
| EXP(x)                          | Exponential function.  |
| FLOOR(a [,kind])                | Greatest integer less than or equal to its argument<br>( <i>kind</i> permitted only in Fortran 95).  |
| HUGE(x)                         | Largest number in the model for numbers like x.  |
| INT(a [,kind])                  | Convert to integer type.   |
| KIND(x)                         | Kind type parameter value.   |
| LEN(string)                     | Character length.  |
| LEN_TRIM(string)                | Length of string without trailing blanks.  |
| LOG(x)                          | Natural (base $e$ logarithm function.  |
| MATMUL(matrix_a, matrix_b)      | Matrix multiplication.   |
| MAX(a1, a2 [,a3,...])           | Maximum value.   |
| MIN(a1, a2 [,a3,...])           | Minimum value.   |
| MOD(a, p)                       | Remainder modulo p, that is $a - \text{int}(a/p) * p$  |
| NINT(a [,kind])                 | Nearest integer.   |
| REAL(a [,kind])                 | Convert to real type.  |
| SIN(x)                          | Sine function.   |
| SINH(x)                         | Hyperbolic sine function.  |
| SIZE(array [,dim])              | Array size.  |
| SQRT(x)                         | Square root function.  |
| TAN(x)                          | Tangent function.  |
| TANH(x)                         | Hyperbolic tangent function.   |
| TINY(x)                         | Smallest positive number in the model for numbers like x.  |
| TRANPOSE(matrix)                | Matrix transpose.  |

### 3 Working with multiple files (optional)

Actually, the program layout given at the beginning of this document is not the whole truth. Similarly to other programming languages, Fortran 90 also offers the possibility of splitting your source code over more than one file. This can be accomplished via two mechanisms: *external subprograms* and *modules*.

**External subprograms:** are located in separate files than the one containing the main program. Consequently, they also have to be compiled separately and then linked to the main program. Please note that the subprograms can also be nested, i.e. an external subprogram can, in turn, call another external subprogram.

Basically, the layout of an external program is very similar to the one of functions and subroutines:

Syntax for **external** subprograms:

```
SUBROUTINE Name ([argument list])
    [subroutine's declaration statements]
    .....[subroutine's executable statements]
    [CONTAINS
        internal subprograms]
END SUBROUTINE [Name]
```

OR:

```
FUNCTION Name ([argument list])
    [function's declaration statements]
    .....[function's executable statements]
    [CONTAINS
        internal subprograms]
END FUNCTION [Name]
```

There are, however, some major differences:

- First of all, whereas the variables and definitions in the main program are accessible inside the internal subroutines (unless you use the INTENT specifiers), they are not accessible from inside the external subprograms. The subprogram relies entirely<sup>10</sup> on the parameters it receives from the calling program unit (most often - main program).
- *External subprograms may contain subprograms of their own*, whereas the internal ones are not allowed to.
- A final (less significant) difference is that the SUBROUTINE and FUNCTION keywords in the headers of the external subprograms are optional, in contrast to the internal subprograms (where they are mandatory).

Let us now re-write the simple program given above that swaps a pair of numbers:

```
file : swap_main.f90
```

```
PROGRAM swap_numbers
IMPLICIT NONE
```

<sup>10</sup>There is, however, a mechanism which allows the definition of global variables and definitions in the case of multiple source files: MODULES. They will be treated in a moment.

```

EXTERNAL swap      ! Required to let the compiler know about
                   ! our external subprogram

REAL :: a=2., b=3.

PRINT*, 'Before_subroutine_call:'
PRINT*, "a=", a
PRINT*, "b=", b
CALL swap(a,b)
PRINT*, 'After_subroutine_call:'
PRINT*, "a=", a
PRINT*, "b=", b

CONTAINS           ! Will NOT compile if we use EXTERNAL!
                   ! This double definition of the swap
                   ! subroutine is an intentional "error",
SUBROUTINE swap(x,y) ! whose purpose is to show you the effect
  REAL x, y, aux    ! of including/omitting the EXTERNAL line.
  x=2*x; y=2*y      ! Try to compile the project by commenting
END SUBROUTINE swap ! either this subroutine definition or
                   ! the EXTERNAL line to see the effects.

END PROGRAM swap_numbers

```

---

end of file: swap\_main.f90

---

file: swap\_subprogram.f90

```

SUBROUTINE swap(x,y)
  REAL x, y, aux
  aux=x; x=y; y=aux
END SUBROUTINE swap

```

---

end of file: swap\_subprogram.f90

Try to compile and run the code. For compiling, you may use the following *Makefile*:

---

file: Makefile

```

OBJS = swap_main.o swap_subprogram.o
F90 = gfortran
.SUFFIXES: .o .f90
.f90.o:
    ${F90} -c ${FFLAGS} *.f90
swap: ${OBJS}
    ${F90} -o $@ ${OBJS}
clean:
    rm -f swap ${OBJS}

```

---

```
end of file: Makefile
```

---

The *Makefile* greatly simplifies the compilation and linking process: to compile this project, change directory from the Linux console to the folder where you stored the three files and type *make swap*<sup>11</sup>.

**The *EXTERNAL* statement:** is not mandatory but constitutes (as the *IMPLICIT NONE* line) a recommended programming guideline. Basically, its purpose is to protect you from unpleasant accidents; if you add by mistake another *internal subroutine*<sup>12</sup> with the same name as the external one, or if you compile the code on another machine that has a built-in subroutine with that name of which you are unaware, the *EXTERNAL* command tells the compiler to discard all other definitions of that subroutine and use only the external one.

**Modules:** have the same basic principle as the previously discussed external datafiles, but with the following differences:

- A module may contain more than one subprogram (referred to as *module subprograms*);
- A module may contain declaration and specification statements which are accessible to all program units which use the module (allowing us to declare global variables/definitions when this is desirable).

The structure of a module is given below:

---

```
Syntax for Fortran modules:
```

---

```
MODULE name
  [declaration statements]
  [CONTAINS
    module's_subprograms]
  END_ [MODULE_ [name]]
```

---

Our number-swapping example could then be written using modules as follows:

---

```
file: swap_main.f90
```

---

```
PROGRAM swap_numbers
  USE my_module      ! Note: If used, the USE command should
                    ! appear first in your source code!
  IMPLICIT NONE
  REAL :: a=2., b=Pi

  PRINT*, 'Before_subroutine_call:'
  PRINT*, "a=", a
  PRINT*, "b=", b
  CALL swap(a,b)
  PRINT*, 'After_subroutine_call:'
  PRINT*, "a=", a
  PRINT*, "b=", b
END PROGRAM swap_numbers
```

---

<sup>11</sup>For a short tutorial on using *make*, see <http://www.wlug.org.nz/MakefileHowto>. The tutorial focuses on integrating *make* with the *C/C++* programming languages, but *make* is really a multi-purpose tool (see the listing above).

<sup>12</sup>As we intentionally did in our listing.

---

end of file: swap\_main.f90

---

file: swap\_module.f90

---

```
MODULE my_module
  REAL, PARAMETER :: Pi=3.141592654
CONTAINS
  SUBROUTINE swap(x,y)
    REAL x, y, aux
    aux=x; x=y; y=aux
  END SUBROUTINE swap
END MODULE my_module
```

---

end of file: swap\_module.f90

with the corresponding *Makefile*:

---

file: Makefile

---

```
OBJS = swap_main.o swap_module.o
F90 = gfortran
.SUFFIXES: .o .f90
.f90.o:
    ${F90} -c ${FFLAGS} *.f90
swap: ${OBJS}
    ${F90} -o $@ ${OBJS}
clean:
    rm -f swap ${OBJS} *.mod
```

---

end of file: Makefile

## 4 Appendix B: Tables

Table 2: Special Characters in Fortran 90

| Special Character | Name              |
|-------------------|-------------------|
| =                 | Equals sign       |
| +                 | Plus sign         |
| -                 | Minus sign        |
| *                 | Asterisk          |
| /                 | Slash             |
| (                 | Left parenthesis  |
| )                 | Right parenthesis |
| ,                 | Comma             |
| .                 | Decimal point     |
| \$                | Currency symbol   |
| '                 | Apostrophe        |
| :                 | Colon             |
|                   | Blank             |
| !                 | Exclamation mark  |
| "                 | Quotation mark    |
| %                 | Percent           |
| &                 | Ampersand         |
| ;                 | Semicolon         |
| <                 | Less than         |
| >                 | Greater than      |
| ?                 | Question mark     |

## References

- [Hahn, 1996] Hahn, B. D. (1996). *Fortran 90 for Scientists & Engineers*. Hodder Headline Group (Arnold).
- [Michael Metcalf, 1999] Michael Metcalf, J. R. (1999). *Fortran 90/95 explained*. Oxford University Press.