Introduction to FORTRAN
A PHY201 Reference Material

Aleksandar Donev

September 1999
Physics Department
State University of Michigan

Contents

Chapter 1

Overview of FORTAN 77

1.1 A Few Logistics

You will receive instructions on how to edit, save, compile your programs etc. along with the class worksheets. However, there are a few things worth noting. We do *not* have a Fortran 90 compiler yet (we are expecting it soon), so you will need to use the Fortran 77 Linux compiler g77, which is simply envoked through:

```
g77 Source_file.f>Executable_file.f
```

The important thing to note is that (fortunately) this compiler supports *some features* (*not all*) that are in the 90 standard, and which are very important to good programing. In particular, it supports the Fortran 90 structure of a DO—END DO loop (along with EXIT and CYCLE), the type declaration statement with the double colon :: syntax, and the standard relational operators instead of the F77 verbose substitutes.

You are expected to read this manual for both the 77 and the 90 standard and use the more modern and accepted syntax whenever possible. The compiler actually recognizes most Fortran 90 commands and gives a message if it does not support it. Hopefully, by the time you come to more advanced programming tasks we will have a Fortran 90 (or even 95) compiler available.

1.2 Program layout

Programs are usually entered using a text editor. It is advisable to follow some guidelines when typing the program:

- Put all Fortran keywords and intrinsic function names in upper case, everything else in lower case.

- All Fortran statements must be contained in column 7 thru 72 of the input file
- Blanks are not significant (except in a character context).
- A statement which is blank in columns 1 to 5 but contains a character other than zero in column 6 is treated as a continuation of the previous non-comment line (a maximum of 19 continuation lines).
- Any statement labels must be written in columns 1 to 5 and consist of up to five consecutive digits.
- Comment lines have a C in column 1
- It is advisable to indent by 2 columns in the body of program units and INTERFACE blocks, DO-loops, IF-blocks, CASE-blocks, etc. for better layout.

The structure of a Fortran 77 program looks like:

```
      PROGRAM name
C Comments and program information
         Declaration of variables and/or external functions
         Body of program...
        END PROGRAM name
        Declaration and body of user-made functions
```

1.3 Variables

It is advisable that all variables be explicitly declared at the beginning of the program. To avoid omissions, include the statement IMPLICIT NONE at the beginning.

1.3.1 Naming Convention

The rules for making names of variables are:

- Names upto 31 characters long:
- Allowed symbols are letters a...z,A...Z, numerals 0,1,2...9, and the underscore _.
- First character must be a letter.
- No case sensitivity.
- Watch for reserved words.

1.3.2 Data Types

In your program, you will use variables of different types, like integers, floating point numbers, strings of letters etc. The types supported in Fortran 77 are:

- Type INTEGER, for integers. The range of integer values is limited, but the exact value depends on the compiler and machine used.
- Type REAL, for floating point numbers (arround 8 digits).
- Type DOUBLE, for floating point numbers with extended precission (about 16 significant digits).
- Type CHARACTER for strings of characters or single characters.
- Type LOGICAL for two-valued boolean variables.
- Type COMPLEX, which is a complex number represented as an ordered pair of REAL's. Most float functions in Fortran 77 can operate on complex numbers as well.

Integer constants are of the form 1234, real constants are of the form 1234.0 or 1.234E3, double precision constants are of the form 1.234D3, and complex numbers are of the form (3.14,-1E5). Characters are enclosed in quotes of the form 'ABab' or 'S'. Logical constants can have only two values .TRUE. or .FALSE. (note the opening and closing periods).

When a numeric variable is too small or too large in magnitude, an under- or overflow results. Division by zero is an error, but the details depend on the system.

1.3.3 Variable Declaration

The compiler needs to now the names, types and sizes of the variables used in the program in order to allocate memory and optimize the performance. All variables must be declared if the IMPLICIT NONE statement is used. Otherwise a type is implied by the first letter:

- a...h and o...z are REAL
- i,j,k,l,m,n are INTEGER

Explicit declaration is done with the statement:
`type variable list [=initial values]`

If we want to assign constant values to some of the variables we make them parameters:
`PARAMETER({List of name=value})`

For numeric variables, type is one of INTEGER, REAL and DOUBLE. We give some of the variables and all of the parameters values at declaration (called initialization).

Examples:

```
INTEGER count, hours, minutes=36
REAL pi
PARAMETER (pi=3.141592)
```

1.4 Basic Expressions

1.4.1 Arithmetic Operators and Expressions

The most important arithmetic operators are addition +, substraction -, multiplication *, division /, and exponentiation **. For example, A + B - C, A*(-B), A*B/C, Z**I.

The order of evaluation is:

1. Parentheses (innermost first)
2. Exponentiation
3. Multiplication and division
4. Addition and subtraction

For example, the expression A*B-C/D is evaluated as if it were written as (A*B)-(C/D). Where two operators have the same priority, the order of evaluation is left to right. For example, A/B*C is evaluated as (A/B)*C which is not equivalent to A/(B*C).

It is important to notice that the result of an operation between floats is a float, between integers is an integer, and between floats and integers is a float. This means that 1/5 gives the integer 0, 1./5. gives 0.2 in single precision, and 1d0/5d0 gives 0.2 in double precision. In general 4 is an integer, so use 4.0 to make it a floating point number.

1.4.2 Assignment statement

The simplest command in any language, including Fortran is the assignment:

`variable = expression`

which assigns the value of the expression to the variable. For example, A=B+C/D**3 is the same as $A = B + \frac{C}{D^3}$.

1.4.3 Relational Operators

Fortran 77 is a special language in that it does not label the standard relational operators like <, >, >=, =< etc. with the usuall mathematical symbols (this is corrected in Fortran 90). In Fortran 77, the relational operators have names between two periods:

.EQ.	equal to
.GE.	greater or equal to
.GT.	greater than
.LE.	less than or equal to
.LT.	less than
.NE.	not equal to

For example, (A .GE. B) is true if the value of the variable A is greater or equal to the value of B. Operators can be applied between mixed types, and as with arithmetic operators, all the arguments are converted to the highest type (integer→real→double→complex). Also, in the 77 standard, character comparison is a machine-dependent operation.

1.4.4 Logical Expressions

You can combine relational expressions and other “true–false”valued expressions and variables together to form logical expressions. This is done using the logical operators:

.NOT. logical negation
This is the only unary operator, the rest are binary:
.AND. logical and
.OR. logical inclusive or
.EQV. logical equivalence
.NEQV. logical non-equivalence (exclusive or)

Since some may not be familiar with boolean logic, here is a boolean table of values for these (T stands for true, F for false):

x	y	.NOT.x	x .AND. y	x .OR. y	x .EQV. y	x .NEQV. y
F	F	T	F	F	T	F
T	F	F	F	T	F	T
F	T	T	F	T	F	T
T	T	F	T	T	T	F

(1.1)

The only rule that you need to know (in case of doubt use excessive bracketing if necessary) is that arithmetic operators take precedence over relational operators which take precedence over logical operators. Thus, these two lines are equivalent (use the second form to be on the safe side):

jill .EQ. jack .OR. boulder .GT. pebble .AND. ant .LT. giraffe
(jill .EQ. jack) .OR. ((boulder .GT. pebble) .AND. (ant .LT. giraffe))

1.5 Characters and Strings

Although strings and characters have been mentioned already and they are not very important in scientific computing, it is essential that some string-operation basics are established, so we devote a separate section to them.

1.5.1 String Declaration

Character strings in Fortran 77 must have a fixed and pre-specified length. This is done in the declaration using *:

```
CHARACTER name*length
```

We can also declare arrays of strings:

```
CHARACTER array(size)*length
```

An exception to this is for string constants (parameters), for which we do not have to count the letters but just put (*) for the length. For example, a printing format specification that we use often in a program might look like,

```
CHARACTER format_spec*(*)
PARAMETER(format_spec='('The date''s',I2,'/',I2,'/',I4)
```

where double apostrophes are used to represent one.

1.5.2 Character Operations

There are two basic operations one can do with strings, extract a part of a string, or join (concatenate) two strings. Extracting a part of a string can be done very easily in Fortran 77 (this type of utility is extended to numeric arrays in Fortran 90), by using the colon : as an ellipse character,

```
substring=string(start position :  end position)
```

where both the start or the end can be omitted if corresponding to the first and last characters. This construction can be used in assignment statements as well.

Concatenation is done using the concatenation operator //. For example,

```
CHARACTER*8 first_word, second_word
First_word='CUPBOARD'
Second_word=First_word(:3)//First_word(4:)
```

makes both the first and the second word 'CUPBOARD'.

1.5.3 Character Functions

There are a number of intrinsic character functions. For example, LEN gives the (declared) length of a string; CHAR and ICHAR perform integer-to-char and char-to-integer conversion; INDEX searches the string for a substring; LGE, LGT, LLE and LLT are relational machine-independent operators for strings, etc. You can learn more about these if and when you need them.

1.6 Input and Output

1.6.1 READ

Most programs require that the user input some data through the keyboard or that the program print some result on the monitor. User input is achieved through the statement READ, with structure:

`READ *,{input list}` or

`READ([UNIT=]unit type,[FMT=]format) {input list}`

The unit type is a number, for example, a 5 for keyboard, and the format type is * for a format-free input. For example,

READ(UNIT=5,FMT=*) a,b,c

expects the user to input three numbers separated with commas.

1.6.2 WRITE

Writing to the screen (or printer) is done using the command WRITE, which can take the form:

`PRINT *,{output list}`

`WRITE([UNIT=unit type],[FMT=]format) {output list}`

The unit type is again a number, usually 6 for the monitor. For example,

WRITE(6,*) ' THE TOTAL IS:', total

prints the message in the quotes and then prints the value of the variable total.

1.6.3 FORMAT Specification

The format can be specified by the user in two different ways. First, the format can be a number giving the label (in columns 1-5 of the program) of the line in the program containing the call,

`FORMAT(format sequence)`

or the format can be a string containing the format sequence, with the syntax

`FMT=('format sequence')`

The format sequnce is a beast of its own. It is a list of *data descriptors* and *control functions*, which specify what type the data that is being printed is and how it should be printed and specify things like new lines.

Data Descriptors

The following table gives the data descriptors for various types of data:

Data type	Data descriptors
Integer	Iw, I$w.m$
Floating point	E$w.d$, E$w.d$Ee, F$w.d$, G$w.d$, G$w.d$Ee
Logical	Lw
Character	A, Aw

(1.2)

Here, the capital letters stand for:

I integers
F a fixed-point floating number
E a float in scientific (exponential) notation
G either F or G, depending on the magnitude of the number
L logical
A character

while the small letters stand for:

w the total filed width
m the minimum number of digits
d number of digits after the decimal point (precision)
e number of digits in the exponent

Control Functions

The most common few control functions are:

/ skip to a new line (record)
'Any string' output a string constant (a message)
nX moves the cursor n columns to the right

For example,

```
      WRITE(UNIT=*,FMT=10) 'The frequency is', f, 'Hz'
10     FORMAT(1X, A, F10.5, A)
```

displays the value of the frequency variable f with a maximum of 15 digits, 5 decimal digits, an indent of 1 column in the beginning, and a suitable string message.

1.6.4 File Input and Output (I/O)

One of the areas in which Fortran 77 (besides number crunching) is better than most other programming languages is file input and output. In most programming languages, file does not refer just to hard-disk files, but to any peripheral device that one can read from or write data to (keyboard, monitor, printer, disks, etc.). Therefore, file I/O is still done using the general READ and WRITE commands (usually), but with a UNIT number that specifies the device or file the data is being read or written to.

OPEN

To assign specific UNIT numbers to certain files and open (or create) the file, use the OPEN statement,

```
OPEN(UNIT=number, FILE='Name of file',STATUS=status, ACCESS=access...)
```

The status can be 'NEW' for new files, 'OLD' for existing files, 'UNKNOWN' if you are not sure or 'SCRATCH' for a temporary file. The access is one of 'SEQUENTIAL' or 'DIRECT'. The distinction is very important, but beginners should use the default of sequential files, which are simply files in which the data are written and read from line-by-line. There are other less-important options.

CLOSE

After we are done using the file, it is very recommended that the file be closed using,

```
CLOSE(UNIT=number, STATUS=status...)
```

where the unit is the number of the file, and the default status is 'KEEP' to save the file or 'DELETE' to delete it (the former is the default).

Implied DO-Loops[1]

A whole array can very efficiently be read or written by simply using their name without subscripts. If you need to write or read a part of an array, implied DO-loops should be used for efficiency (the same goes for WRITE):

```
READ(UNIT=number,FMT=format)({data list}, loop variable=start, limit,
step)
```

This is equivalent to the reading or writing statement with the usuall format and data list operands being placed inside a DO-loop with the loop variable going from the start to the limit in step increments.

For example, to write our income for all months in 1996 to the file "Income_1996.dat", we would type:

```
OPEN(UNIT=13,FILE='Income_1996.dat',STATUS='NEW')
WRITE (UNIT=13, FMT='(I2,F10.2,/)') (month,income(month,1996),month=1,12,1)
```

[1] Come back to this section after becoming familiar with DO loops.

CLOSE(UNIT=13)

1.7 Functions and Subroutines

1.7.1 Intrinsic Functions

There is a library of intrinsic functions available to any Fortran program. These functions are invoked by using the function name followed by its parenthesized list of parameters:

`function name({list of parameters})`

A function is said to return a value based on the values passed to it through the arguments. Some of the more important intrinsic functions are ABS, ACOS, COS, DOT_PRODUCT, EXP, INT, LEN, LOG, LOG10, MATMUL, MAX, MIN, MOD, NEAREST, NINT, REAL, SIN, SQRT, TAN, TRANSPOSE. See Appendix A for a fuller list and details.

Many intrinsic functions are generic in the sense that the function may be used with different (but not mixed) data types as parameters to the function. For example, ABS, COS, MAX, MIN. Some functions though, accept only certain types of arguments, for example, most of the the float-valued functions, like SQRT, SIN, LOG etc. Thus, SQRT(4) is invalid! Use SQRT(4.0) instead. Some functions have different versions for different types of arguments (see Appendix A), and the generic function calls these specialized functions depending on the types of the arguments. For example, to calculate the square root of a given integer or any floating number (even complex), we simply write root=SQRT(1.0*number).

1.7.2 Intrinsic Subroutines

Subroutines are very similar to functions, but there is an important distinction. Functions return one value and are not recommended to change the values of their parameters. Subroutines do not return a value explicitly, but execute a well defined group of statements (activity) and can freely change the values of their parameters. They should also be used when we wish to return more than one values.

Subroutines are invoked using a CALL statement:

`CALL name({list of arguments})`

Some of the intrinsic subroutines include DATE_AND_TIME, MVBITS, RANDOM_NUMBER, RANDOM_SEED, SYSTEM_CLOCK etc.

1.7.3 External Functions

You can define your own functions, usually after the END of the program. The layout for functions is:

`type FUNCTION name ({dummy arguments})`

```
  local variable declaration
  body of function...
  name = expression
  body of function continued if needed...
END FUNCTION name
```

The type of the name identifies the type of the result that will be returned by the function. The result of the function is the value of the function that is assigned to the name of the function within the body of the function. The function therefore **must** contain at least one assigment of a value to the name of the function.

The *dummy arguments* are a list of constants, variables and even procedures which are accessible from within the body of the function.When the function is used, it will have corresponding *actual arguments* that must be of the same type and length as the dummy arguments, but not necessarily the same names. No type checking is done during compilation or run time, and you will get very weird errors if the types don't match. All arguments are passed using "call by reference" (like VAR arg in Pascal or &arg in C++). Changing the value of an argument in the subroutine changes the value of the corresponding variable in the calling program.

For example, to define a function that calculates the gravitational force between two bodies, we define a function Newton:

```
REAL FUNCTION Newton (m1,m2,r)
  REAL gamma=6.672E(-11), m1, m2, r
  Newton=-gamma*m1*m2/r**2
END Newton
```

1.7.4 Statement Functions

When the function you are implementing is a "one-liner", then you can save typing by defining the function in a single line:

```
function_name({list of parameters})=expression
```

For example, we can calculate the force of graviational interaction from Newton's law with (note that in this case we can use the variable gamma defined elsewhere):

```
NEWTON(m1,m2,r)=-gamma*m1*m2/r**2
```

1.7.5 External Subroutines

The structure od a subroutine is:

```
SUBROUTINE name {dummy argument list}
  local variable declaration
  body of subroutine...
```

```
END SUBROUTINE name
```

Subroutines are accessed by using the CALL statement, and are in most respects similar to functions. When a subroutine is called the dummy arguments in the subroutine become alias names for the actual arguments in the calling statement, i.e. they represent the same physical location in memory. Thus, if the dummy arguments are modified within the subroutine, then so are the actual arguments in the calling statement.

RETURN, SAVE, EXTERNAL and INTERNAL

All well defined functions should have a single-point entry and a single-point exit, but in some situations there may be a need to terminate a procedure (in case of error, let's say). This is done by simply calling RETURN, which stops the execution of the function and returns control to the calling routine.

After the function is finished, the values of the local variables are lost. If they are to be used in a later call of the function, this should be made clear by saving the values of these between function calls (static allocation):

```
SAVE [{list of values to be saved}]
```

When a function is passed on as an argument to another function, and in some circumstances when we wish to link a named block data subprogram into the final executable, the function has to be declared as either intrinsic or external, via the statements:

```
INTERNAL {list of function names}
EXTERNAL {list of function names}
```

For example, here is how to make a subroutine Graph that plots a function in the interval [0,1.0]:

```
      SUBROUTINE Graph(my_function)
        INTEGER i
        REAL x
        DO 10, x=0,1.0,1.0/100
        CALL PLOT(x,my_func(x))
10    CONTINUE
      END Graph
```

Now, if we wish to plot the sine function, we write:

```
INTRINSIC SIN
CALL Graph(SIN)
```

1.8 Control Structures

Control structures determine the program flow—the sequence of execution of the commands (other than the default ordering in the program file). In Fortran 77, these are:

1.8.1 IF-Blocks

An IF block is a multi-branched control structure which takes the execution to different branches depending on the value of one or several condition statements. It's general structure is:

```
IF(First condition statement) THEN
 First sequence of commands
ELSE IF (Second condition statement) THEN
 Second sequence of commands
ELSE IF...
...
ELSE
 Alternative sequence of commands
END IF
```

Any but the first IF can be omitted. The n^{th} (ELSE) IF sequence of statements is evaluated if its condition statement is true and if none of the preceeding conditions were true. In other words, once a condition statement is found true, its sequence of commands is evaluated and the IF-block is exited. The ELSE sequence of commands is evaluated if none of the condition statements are true.

For example, to find the sign of a number ($signum(x) = 1$ for $x > 0$, -1 for $x < 0$ and 0 for $x = 0$) we can use the following construction:

```
IF (number.LT.0)
 signum=-1
ELSE IF (number.GT.0)
 signum=1
ELSE
 signum=0
END IF
```

Logical-IF Statement

Again, many IF statements are one liners and more clarity and less typing are achieved using an abbreviated IF one-liner:

```
IF(Condition statement) Statement to be executed
```

which simply omits the THEN and END IF in the usual conditional IF-block.

1.8.2 DO-Loops

One of the biggest downsides of Fortran 77 is the scarcity of appropriate (infinite) repetition structures and the existance and wide use of the GO TO statement. These have been corrected in Fortran 90, but in 77 the main repetition control sequence is the DO-loop, with structure:

```
DO label, counter=start, limit, step
  body of loop
```

`label CONTINUE`

The label is a number which labels the position of the last statement in the DO loop–usually the CONTINUE dummy statement. It must be placed again in front of the CONTINUE in *columns 1-5* of the program file. The counter is a variable that is automatically incremented by the step from the start until the limit is reached. If a step is missing, 1 is implied. The program flow can freely be taken out of both the DO-loop and the IF-statement, but not into them.

For example, to sum the odd integers from 1 to n, we write:

```
      INTEGER sum=0,i
      DO 10, i=1,n,2
        sum=sum+i
10     CONTINUE
```

1.8.3 GO TO Statement

The GO TO statement is now considered bad programming, but it is an important component of Fortran 77. There are two types of GO TO statements:

Unconditional GO TO

When we want to transfer the program flow to a part of the program, we use the statement,

`GO TO label`

where the label is again a number placed in columns 1-5 of the statement we wish to jump to. For example, to have the user input numbers until he inputs 0 we can use the following semi-indefinite loop:

```
10    READ (UNIT=*,FMT=*), a
      IF (a.NE.0) GO TO 10
```

Conditional (Computed) GO TO

When there a large number of flow branches depending on the value of an integer variable, we use the computer GO TO (`case` in C++):

`GO TO (First label,...., ` n^{th} ` label), Integer-valued expression`

The effect of this is that control is passed to the line with the n^{th} label if the value of the integer expression is n. Note that this construction should be avoided.

1.8.4 STOP Statement

If we want to stop the program and return the control to the operating system, we can use the command:

`STOP ['Message']`

The message is usually displayed on the screen and is optional.

1.9 Arrays

Arrays are collections of data of the same type. They are the most important data type in scientific computing where we usually deal with a great number of, let's say, data points.

1.9.1 Declaration of Arrays

Arrays are declared in the same way as ordinary variables, only by specifiying the starting and ending indices (subscripts):

`type array({Lower bound in` n^{th} `dimension:Upper bound in` n^{th} `dimension})`

For example, to declare an array used to store your income for several years and all the months in each year, use either of these:

```
REAL income(12,1995:1999)
REAL income(1:12,1995:1999)
```

The lower bounds are optional and default to one. The arrays can have up to 7 dimensions. The size of the array must be determined at compilation time. They are stored in contingent blocks of memory and are column-ordered (this is important when exploring an array with nested DO-loops—put the column loop inside the row loop to increase speed).

1.9.2 Using Arrays

Arrays are usually accessed on an element-by-element basis (at least in this old version of Fortran). To access a specific element of the array, simply enclose the list of subscripts for that element in parenthesis:

`array({List of subscripts})`

For example, to find your july income in 1996, use:

```
july_income=income(6,1996)
```

In certain occassions arrays can be used as a whole, without subscripts. The most important are when printing or reading an array or passing the array to a function as an argument. The first case is trivial. For example, to print a whole array, just use:

```
WRITE(5,*) array
```

The second case is quite tricky in Fortran 77.

1.9.3 Arrays as Arguments to Functions

Passing arrays to functions is not one of the highlights of Fortran 77, and many improvements exist in Fortran 90. The main problem is that the arrays contain no information about their own size, so that the size of the arrays passed as arguments to functions have to be either *fixed-size arrays*, or the size of the arrays has to actually be passed to the function as an extra argument—*adjustible arrays*. If the size of the array is unknown, than it does not have to be specified, and we can use *assumed-size arrays*. We declare this with an * as the size of the last dimension (other dimensions can not be assumed). But remeber that the compiler has no information about the size of the array and so we must ensure that we stay within the bounds of the array. Also, this excludes using the arrays in READ and WRITE statements without subscripts.

For example, to define a function that calculates the norm of a vector (a one dimensional array), we would type:

```
      REAL FUNCTION Norm(array,array_start,array_end)
        REAL sum=0.0,array(array_start,array_end)
        INTEGER i
        DO 10, i=array_start,array_end
        sum=sum+array(i)**2
10      CONTINUE
        Norm=SQRT(sum)
      END Norm
```

Chapter 2

Overview of New Features in FORTAN 90

In this overview we do not repeat the constructs that have not been changed from the 77 standard. Most Fortran 90 (or higher) compilers are fully backward compatible, meaning that they can compile purely 77 code as well. However, it is advisable to switch to the new syntax because of its many advantages.

It is quite worthwhile saying a few things about the type of improvements made in Fortran 90. Loosely speaking, two major types of improvements have been made:

- Improvements to make Fortran a more flexible and powerful programming language, with the possibility of object-oriented and well-structured programming, access to more system-level processes like memory allocation and pointers, etc. In other words, Fortran 90 looks a lot more like C++ than the 77 standard. Fortran still has some ways to go in this respect.
- Improvements to improve numerical capabilities, in the first place by incoorporating parallel (vectorized) syntax commands. This is a major improvement in terms of scientific computing and allows the writing of "universal" parallel code. More has been done on this in the 95 standard (the 2000 standard is not expected to be publicly available for another 2 years), and we will discuss this in more detail later.

2.1 Program layout

The program structure is almost identical to the Fortran 77 structure, with a few improvements to achive compatibility with new text editors:

- A line may contain up to 132 characters and may contain more than one Fortran statement provided a semicolon separates each successive pair of statements.
- Blanks are significant.
- A trailing ampersand & indicates a statement is continued on the next line (a maximum of 39 continuation lines). Note that comments therefore cannot be continued, and a separate exclamation mark (!) must be used for each comment line.
- Statement labels consist of up to five consecutive digits preceeding the command.
- Comment lines must begin with the exclamation mark !. Also, trailing comments (after a command) are allowed and also go after !.

The structure of a Fortran 90 program looks like:

```
PROGRAM name
!Comments and program information
 Declaration of variables and/or external functions
 Body of program...
[CONTAINS
Internal procedures]
END PROGRAM name
Declaration and body of user-made functions
```

2.2 Variables

2.2.1 Variable Declaration

Explicit declaration is done with the slightly modified statement:

```
type [, attributes] ::  {variable [=initial values]}
```

If we want to assign constant values to some of the variables we make them parameters by making PARAMETER one of the attributes. Also, the length of a character string can be specified via the LEN parameter:

Examples:

```
INTEGER :: count, hours, minutes=36
REAL, PARAMETER :: pi=3.141592
CHARACTER (LEN=10) :: name, surname
```

2.2.2 Data Types

Fortran 90 attempts to make code more transferable and architecture-independent by allowing the user to specify the length (in words) of the variables through the KIND attribute (note that this keyword can be ommitted):

```
type([KIND=]kind_num) [, attributes] :: {variable list}
```

The problem is that a word is defined differently on different machines. On most architectures (like the PC), for example, a REAL with 8 words is double precision, and with 4 words is a single precision float:

```
REAL(8) :: double_number=2.0D3
```

A good programmer will make one separate record (module) for each of the machines he runs programs on, and make symbolic names for the kinds that will be universal, while changing the length depending on the machine architecture.

2.2.3 Derived Data Types

Derived data types are an equivalent to C's structures, and are groups of data types that are always used together. The syntax is,

```
TYPE name
  Variable specifications
END TYPE name
```

and the new data type is referenced as TYPE(name). This will be easier to explain with an example. Let's assume that we want to write programs that use vectors, and that Fortran does not support them. So, we define a new data type for two dimensional vectors:

```
TYPE vector
  REAL :: x_component, y_component
END TYPE vector
```

Now we can declare two vectors, assign their numerical values, and form their sum by accessing the individual elements with %:

```
TYPE(vector) :: a, b, sum
a=vector(1.1,2.5)
b=vector(2.0,5.0)
sum=vector(a%x_component+b%x_component,a%y_component+b%y_component)
```

2.3 Functions and Subroutines

Many changes have been made in Fortran 90 concerning functions and subroutines (procedures), arrays and control structures. Some of these changes are more advanced, so we will not go into many details. It is instructive to explain the structure of a Fortran program once again.

2.3.1 Program Units

Each program consists of program units, called *scoping units.* These can be the main program, subprograms, procedures, etc. The important thing to notice is that each scoping unit is independent in terms of its variable space, so that it is good to try to put well defined and more or less isolated groups of statements into separate units. This way, you can invoke that unit from various programs, without worrying about possible variable conflicts.

Any unit that is placed (defined) within another unit is a *sub-unit* and has access to all the variables in the unit it is a part of. Therefore, procedures that interact with the main program variables should be placed inside the PROGRAM, using the CONTAINS command. These are *internal procedures*, as contrasted to *external procedures*, which are defined outside of the PROGRAM unit and have their own variable workspace. The communication, or *interface*, between the main program or subprogram and the external procedure is done throught the argument list. Data is copied from actual arguments to dummy arguments upon the invocation of the procedure (this is not always true), and from dummy argument to actual argument upon exiting the procedure.

Another way of sharing of data between procedures is the usage of modules, which are new units introduced in Fortran 90, facilitating global variables and procedures. We now describe in more detail some of the new feautures of the 90 standard.

2.3.2 Internal Procedures

Internal procedures should be placed after the CONTAINS command. They can be a part of any program unit, like the main program, a subprogram or a procedure (usually no more than 2-3 levels of nesting). For example, here is how to make a procedure that quickly zeros all the integer counters that we use in the main program:

```
PROGRAM counters
  INTEGER :: i, j, k, l, m, n
  CALL Zero_counters ! Zero all the counters
  CONTAINS
    SUBROUTINE Zero_counters
      i=0; j=0; k=0; l=0; m=0; n=0;
    END SUBROUTINE Zero_counters
END PROGRAM counters
```

2.3.3 External Procedures

As already explained, external procedures are usually placed outside the main program (in a separate file, a module, the same file, etc.). When the compiler compiles the program, it can detect a great deal more errors if it is told what the procedure's interface (argument list) is. This is done with the INTERFACE block, which contains the declarations of the external functions (it can be placed inside the main program or in a separate module).

Also, the compiler can be told what the intention of the arguments of a subroutine is, using the INTENT data attribute with one of:

- IN for arguments that should not be altered in the subroutine, but only pass data to the procedure
- OUT for arguments that need to be assigned values in the subroutine
- INOUT for arguments that pass data both in and out of the procedure

For example, here is an interface to a subroutine that returns the time in seconds given the time in hours, minutes and seconds:

```
INTERFACE
  SUBROUTINE convert_time(hour,minute,second,time)
    INTEGER, INTENT(IN) hour, minute, second
    INTEGER, INTENT(OUT) time
  END SUBROUTINE convert_time
END INTERFACE
```

2.3.4 Modules

Modules are a new program unit in Fortran 90. They are mostly used to enclose data (global variables) and INTERFACE declarations of functions that several other program units need to share. One can also put full procedure bodies inside a module (so called module programs). The module is defined via:

```
MODULE name
  Global variables declarations
  INTERFACE blocks
  ...
END MODULE name
```

If we want a program unit to have access to the variables and procedures defined in the module, we use the USE command. If we have lots of procedure interfaces in the module (for example, a subroutine library) but would want to use only a few of these procedures, we specify this with the ONLY attribute (this is also a way to make some global variables not accessible to all program units and is good programming practice):

```
USE module name [, ONLY: {procedure names}]
```

For example, we can store the total number of floating-point operations (flops) in a program in a module,

```
MODULE flops
  INTEGER(KIND=8) :: flops_count
END MODULE flops
```

and then update this number from any program unit that contains the statement:

```
USE flops
```

2.4 Control Structures

Control structures have not changed a lot in Fortran 90. We describe one new C-like branching stattement and some important additions to the repetition statement. One important addition is that control structures can be named with a descriptive name which preceeds the control structure:

```
[name:]  Control Structure
```

2.4.1 CASE-Blocks

The CASE structure is similar to the computed GO TO statement. It is used when the program flow needs to be branched in several different directions depending on the value of an expression. The syntax is:

```
[name:]  SELECT CASE (expression)
  CASE (value) [first name]
    First block of statements
  ...
  [CASE DEFAULT
    Default block of statements]
END SELECT [name]
```

A limitation to this construct is that the result of the expression may be character, integer or logical only. The value specification may take single value(s) or a range of values separated by a colon : (henceforth called an *ellipse*). For example, here is a CASE structure that determines which season a month is in:

```
INTEGER :: month
CHARACTER*10 :: season
Seasons: SELECT CASE(month)
  CASE(4,5)
   season='Spring'
  CASE(6,7)
    season= 'Summer'
  CASE(8:10)
    season= 'Autumn'
  CASE(11,1:3,12)
```

```
    season= 'Winter'
  CASE DEFAULT
    season='Invalid month'
END SELECT Seasons
```

2.4.2 DO-Loops

It was said in the first chapter that in Fortran 77 the GO TO statement is needed to make infinite loops. Since the GO TO statement is today considered very bad programming, the 90 standard offers a kind of an improvement (it is unclear why there is no DO WHILE statement in Fortran) by using named do loops and the two new commands, EXIT and CYCLE, whose syntax is,

```
EXIT [name of loop]
CYCLE [name of loop]
```

and they either exit a given loop (the innermost DO loop by default), or start another iteration. To make an infinite loop, one can thus use an "empty" named DO loop and EXIT to stop the iteration. For example, here is a program segment in which the user inputs an array of 5 numbers and the computer calculated their inverse square roots, so long as none of the numbers is negative:

```
REAL :: array(5)
outer: DO
  READ(*,*) array
    inner: DO k=1,5
    IF (array(k)==0) CYCLE inner
    IF (array(k)<0) EXIT outer
    array(k)=1/SQRT(array(k))
  END DO inner
WRITE(*,*) array
END DO outer
```

It should be noted however that named loops have the same negative potential as the GO TO statement and should be used with caution.

2.5 Arrays

Some of the most important changes in Fortran 90 come from the changes concerning arrays and array operations. We thus carefully explain the importance of these changes, although some topics like pointers are too complicated to explain in detail.

2.5.1 Declaration of Arrays

Arrays are declared in the same way as in the 77 standard, with the important inclusion of several new attributes:

`type [,attributes] array({Lower bound:Upper bound})`

The more important attributes are

- `DIMENSION({Lower bound:Upper bound})`, which enables that the dimension be specified, and the lower and upper cound can now be actual workspace variables, thus enabling the existence of *automatic arrays* in procedures, which come to existence with different dimensions each time the procedure is called. Also, it is very important to note that the dimensions do not have to be specified at compile time, in which case they are substituted with the ellipse symbol. If the dimensions of an array argument to a procedure are not specified, the procedure must have an INTERFACE block in the calling unit, so that the compiler knows that the array has to be passed along with its dimensions.
- ALLOCATABLE, which identifies the array as allocatable.
- TARGET, which identifies the array as a target for pointers.

For example, to declare an array used to store your income for several years and all the months in each year, one can use:

REAL, DIMENSION(12,1995:1999) :: income

2.5.2 Vectors

Vectors are one-dimensional arrays which is used mainly to access sub-arrays of a given array (see the next subsection). It deffers from an array only in the way it is assigned a value by using constructors:

`vector = (/{[lower:upper:step],.../)`

Where list may be a list of values of the appropriate type, namely, a variable expression, array expression, implied DO loop or any combination of these. This will be clearer later on, but here is an example of a vector that has the value (/ 1 3 5 7 9 6 2 4 8 16/):

INTEGER :: vector(6)=(/1:9:2,6,(2**i,i=1,4)/)

2.5.3 Using Arrays

Arrays in Fortran 90 can be accesses on an element-by-element basis, just like in the 77 standard, but a major improvement is that whole sections of the array can be accesses using one of the two following constructs::

`array({start:end:step})`

`array(vector)`

Any of the starting, ending and step values for the indices can be omitted, so long as there is one ellipse. How this works will become clear only with examples:

```
REAL :: array(50,50), vector(3)=(/1 7 37/)
! This accesses the section of the array from rows 1-20
! and columns 5-10, thus returning an array of size [20,5]
WRITE(*,*) array(1:20,5:10)
! This accesses all the even-numbered rows
WRITE(*,*) array(2:50:2,:)
! And this accesses the elements at the intersection of
! rows 1, 7 and 37 with columns 1, 7 and 37
WRITE(*,*) array(vector,vector)
```

2.5.4 Array Operations

A very important new feature in Fortran 90 is the idea of array-array operations betweeen *conformable arrays.* Two arrays are said to be conformable if they have the same size (not neccessarily the same row or column numbering), or if one of the two arrays is a scalar. *An operation between two conformal arrays is carried between the elements of the two arrays individually.* For example,

```
REAL, DIMENSION(20,20) :: A, B, C
C=A+B
```

assigns to each element in C the value of the sum of the corresponding elements of A and B. Thus, you may say that this is equivalent to a DO loop:

```
DO i=1,20
  DO j=1,20
    C(i,j)=A(i,j)+B(i,j)
  END DO
END DO
```

But this is not quite true, and it is important to understand the difference. The matrix statement C=A+B *does not span in time—it spans in space.* In other words, as far as summing two matrices goes, the order in which the summation is done is arbitrary. On the other hand, the double DO-loop structure *spans in time, not in space.* Therefore, the double loop is an un-natural translation of the matrix statement. The reason that nested looping is the traditional way of performing matrix operations is that most of the computers people commonly use are so called von Neuman sequential machines, in which there is a single processor that does things in a time ordered fashion. However, today the concept of parallel, or multi-processor machines is an essential one, so that nested do loops should be avoided whenever an equivalent array-array (vectorized) statement exists.

When the compiler sees a statement C=A+B for matrices, it automatically optimizes the process for the specific machine using the fact that the operation is not sequential. How it does that is fortunately not our concern—It may use several processors to do that if the machine has more than one, it may access the elements column or row-wise, use pointers (those that are more experienced know that one of the fastest operations is the incrementation of a pointer, not an integer, by one, so that an array can be accessed most quickly with a sweeping pointer), etc. The lesson from all this is that you should always try to formulate your code in vectorized statements.

2.5.5 Elemental Functions

To continue the previous discussion, let's assume we need to take the sine of all the elemens of the array A. Again, this is not a time-ordered operation, and Fortran 90 offers a very efficient way of doing this. We can simply use SIN(A) to perform this operation, because SIN, like most other numerical functions in Fortran 90, is an *elemental procedure*–acting on each element of an array.

Thus a statement like,

```
C(5:10,:)=SIN(A(2:10:2,:))+2.0*B(5:10,:)
```

performs a very complicated matrix operation of assigning to the 5-10$^{\text{th}}$ rows of C the sum of the sine of the elements of A in the even rows of A up to the 10$^{\text{th}}$ row and the doubled value of the elements of the matrix B in columns 5-10$^{\text{th}}$.

2.5.6 WHERE

The WHERE construct is a very useful control structure for performing an operation only on certain elements of an array that satisfy a given condition. The structure is:

```
WHERE (array condition)
 Body of structure
END WHERE
```

For example,

```
REAL, DIMENSION(20,20) :: A, B, C
WHERE (B>0.0)
  C=A/B
END WHERE
```

assigns each element of the matrix C the quotient of the corresponding elements of matrices A and B, so long as the element of B is non-zero.

It should be noted again that the WHERE construct is not equivalent to a DO-loop with a nested IF statement, since a WHERE loop is not time-ordered and can be performed in parallel. A major limitation of Fortran 90 is that WHERE constructs can not be nested. This is corrected in Fortran 95, where WHERE and FORALL structures can be nested at any level.

2.5.7 FORALL (Fortran 95)

We only briefly mention the new (and most important) control structure in Fortran 95, the FORALL structure. This is a space-spanning equivalent to nested DO-loops that performs complicated operations whose time execution must be arbitrary. For example,

```
FORALL (i=1:100,i=1:100, ( i>=j .AND. (x(i,j).NE.0) ) )
y(i,j)=1.0/x(i,j)+x(j,j)**2
END FORALL
```

performs a more sophisticated matrix-like operation. Note that we could not have written,

```
x(i,j)=1.0/x(i,j)+x(j,j)**2
```

since the result would depend on the time ordering of the operations.

Other important improvements in Fortran 95 are the introduction of user-defined PURE and ELEMENTAL function, but the changes are not very important at this level.

2.5.8 Array Intrinsic Functions

With all the changes to the concept of arrays, Fortran 90 contains a lot of new intrinsic functions for arrays. It is beyound the scope of this text to explain these. Here is just a brief list of some of them and what they do, so you know what to look for when you need them:

- `SIZE(array,[dimension])` is probably the most important inquiry funtion for array which finds the rank of an array along the specified dimension (or total number of elements). Use `SHAPE(array)` to get the shape of the array as an array of integer ranks.
- `SUM(array,[dimension])` and `PRODUCT(array,[dimension])` return the sum or product of the elements of an array along the specified dimension.
- `MINVAL(array,[dimension])` and `MAXVAL(array,[dimension])` give the smallest and maximum value of an array along the specified dimension.

- `MATMUL(first array, second array)` is a very important function that returns the matrix product of two arrays (optimized for parallelization when possible).
- `DOT_PRODUCT(first vector, second vector)` finds the dot product of two vectors.
- `RESHAPE(array,shape)` reshapes an array to the ranks (dimensions) specified in the shape array.
- `TRANSPOSE(matrix)` gives the transpose of a two dimensional matrix.

2.5.9 Allocatable Arrays

An essential improvement in Fortran 90 is memory menagement and the introduction of allocatable arrays, specified with the ALLOCATABLE attribute. These arrays do not have a specified dimension and get allocated or deallocated using the commands:

```
ALLOCATE(array({dimensions}))
DEALLOCATE(array)
```

For example,

```
REAL, DIMENSION(:,:), ALLOCATABLE :: array
ALLOCATE(array(10,40))
DEALLOCATE(array)
```

first reserves memory for a 10 by 40 array, and then frees the memory to the memory pool.

2.5.10 Pointers

The concept of pointers is difficult to explain and the true nature of pointers is still not captured well enough in the 90 standard (as it is in C, where pointers are a central feature). For the purposes of this text, a pointer can be understood as a variable that points to a memory location (or locations). In particular, a pointer can point to a block of memory that we want to use to store data, or, more importantly, it can point to the memory location of another variable, called the *target* of the pointer. A pointer is assigned a target via:

```
pointer=>target variable or array
```

When a pointer is declared, it is born in an *undefined* status, when it is assigned a target it becomes *associated*, and to avoid inadvertent misuse we can make it *disassociated* with:

```
NULLIFY(pointer)
```

For example,

```
REAL, DIMENSION(:), POINTER :: one_row
REAL, DIMENSION(50,50), TARGET :: array
one_row=>array(5,:)
NULLIFY(one_row)
```

associated the pointer to the 5th row of the array. This is a useful construction because it is more efficient and easy to manipulate, then let's say, keeping in memory the number 5 to know which row we want to reference.