INTRODUÇÃO À PROGRAMAÇÃO EM FORTRAN 90

Resumo do livro

FORTRAN 90 PROGRAMMING
T. M. R. Ellis, Ivor R. Philips, Thomas M. Lahey
Addison-Wesley: 1994

F. J. Romeiras / 3.NOV.2003

CONTENTS

Chapter 1.	Introduction	1
Chapter 2.	First steps in Fortran 90 programming	3
Chapter 3.	Essential data handling	9
Chapter 4.	Basic building blocks	19
Chapter 5.	Controlling the flow of your program	31
Chapter 6.	Repeating parts of your program	39
Chapter 7.	An introduction to arrays	47
Chapter 8.	More control over input and output	55
Chapter 9.	Using files to preserve data	63
Chapter 10.	An introduction to numerical methods in Fortran 90 programs	70
Chapter 13.	Array processing and matrix manipulation	77

CHAPTER 1. INTRODUCTION

OVERVIEW

Computers are today used to solve an almost unimaginable range of problems, and yet their basic structure has hardly changed in 40 years. They have become faster and more powerful, as well as smaller and cheaper, but the key to this change in the role that they play is due almost entirely to the developments in the programming languages which control their every action.

Fortran 90 is the latest version of the world's oldest high-level programming language, and is designed to provide better facilities for the solution of scientific and technological problems and to provide a firm base for further developments to meet the needs of the last years of the 20th century and of the early 21st.

This chapter explains the background to both Fortran 90 and its predecessor, FORTRAN 77, and emphasizes the importance of the new language for the future development of scientific, technological and numerical computation. It also establishes certain fundamental concepts, common to all computers, which will provide the basis for further discussion in later chapters.

SUMMARY

- **Programming languages** are used to define a problem and to specify the method of its solution in terms that can be understood by a computer system.
- A **high-level language** enables a programmer to write a program without needing to know much about the details of the computer itself.
- Hundreds of programming languages have been developed over the last fifty years. Many of these are little used, but there are a small number which are very widely used throughout the world and have been standardized (either through formal international processes or as a result of *de facto* widespread acceptance) to encourage their continuing use. Most of these major languages are particulary suited to a particular class of problems, although this class is often very wide.
- Two languages stand head and shoulders above the others in terms of their total usage. These languages are COBOL (first released in 1960) and Fortran (first released in 1957). COBOL is used for business data processing and it has been estimated that over 70% of all programming carried out in 1990 used COBOL! Fortran programs probably constitute around 60% of the remainder, with all the other languages trailing far behind.
- Fortran was originally designed with scientific and engineering users in mind, and during its first 30 years it has completely dominated this area of programming.
- Fortran has also been the dominant computer language for engineering and scientific applications in academic circles and has been widely used in other, less obvious areas, such as musicology, for example. One of the most widely used programs in both British and American Universities is SPSS (Statistical Package for the Social Sciences) which enables social scientists to analyse survey or other research data. Indeed, because of the extremely widespread use of Fortran in higher education and industry, many standard **libraries** have been written in Fortran in order to enable

programmers to utilize the experience and expertise of others when writing their own Fortran programs. Two notable examples are the IMSL and NAG libraries, both of which are large and extremely comprehensive collections of **subprograms** for numerical analysis applications. Thus, because of the widespread use of Fortran over a period of more than 30 years, a vast body of experience is available in the form of existing Fortran programs.

- **Fortran 90** is the latest version[1] of the Fortran language and provides a great many more features than its predecessors to assist the programmer in writing programs to solve problems of a scientific, technological or computational nature. Some of these new features were based on the experience gained with similar concepts in other, newer, languages; others were Fortran's own contribution to the development of new programming concepts. Fortran 90 contains all the modern features necessary to enable programs to be properly designed and written – which its predecessor, FORTRAN 77, did not.
- Fortran 90 retains all of FORTRAN 77, that is, any standard FORTRAN 77 program or procedure is a valid Fortran 90 program or procedure, and should behave in an identical manner. Thus all the wealth of existing Fortran code, written in accord with the FORTRAN 77 standard, can continue to be utilized for as long as necessary without the need for modification. Indeed, it is precisely this care for the protection of existing investment that explains why Fortran, which is the oldest of all current programming languages, is still by far the most widely used language for scientific programming.
- Fortran 90 has, therefore, given a new lease of life to the oldest of all programming languages, and is already being used as the base from which still more versions of the language are being developed. The ability to write programs in Fortran 90 will undoubtedly, therefore, be a major requirement for a high proportion of scientific and technological computing in the future, just as the ability to use FORTRAN 77, and before that FORTRAN IV, was in the past.
- Definitive stages in the development of Fortran:

FORTRAN, 1957	IBM Mathematical **FOR**mula **TRAN**slation System, was developed at IBM to provide a more efficient and economical method of programming its 704 computer than the machine code used at that time.
FORTRAN II, 1958	An improved version of the language, with a considerably enhanced diagnostic capability and a number of significant extensions.
FORTRAN IV, 1962	A further improved version almost totally independent of the computer on which the programs were to be run.
FORTRAN 66, 1966	American Fortran standard (American National Standards Institute, ANSI, 1966)
FORTRAN 77, 1978	American Fortran standard (ANSI, 1978)
Fortran 90, 1991	The emergence of Fortran as a modern computer language (ISO/IEC, 1991)

[1]Fortran 95, adopted in 1997 (ISO/IEC, 1997), is a minor revision of Fortran 90 and backward compatible with it, apart from a change in the definition of an intrinsic function and the deletion of some Fortran 77 features declared obsolete in Fortran 90.

CHAPTER 2. FIRST STEPS IN FORTRAN 90 PROGRAMMING

OVERVIEW

The most important aspect of programming is undoubtedly its design, while the next most important is the thorough testing of the program. The actual coding of the program, important though it is, is relatively straightforward by comparison.

This chapter discusses some of the most important principles of program design and introduces a technique, known as a structure plan, for helping to create well-designed programs. This technique is illustrated by reference to a simple problem, a Fortran 90 solution for which is used to introduce some of the fundamental concepts of Fortran 90 programs.

Some of the key aspects of program testing are also briefly discussed, although space does not permit a full coverage of this important aspect of programming. We will return to this topic in the intermission between Parts I and II of this book.

Finally, the difference between the old fixed form way of writing Fortran programs, which owed its origin to punched cards, and the alternative free form approach introduced in Fortran 90 is presented. Only the new form will be used in this book, but the older form is also perfectly acceptable, although not very desirable in new programs.

SUMMARY

- Programming is nowadays recognized to be an **engineering discipline**. As with any other branch of engineering it involves both the learning of the theory and the incorporation of that theory into practical work. In particular, it is impossible to learn to write programs without plenty of practical experience, and it is also impossible to learn to write good programs without the opportunity to see and examine other people's programs.
- The reason for writing a program, *any program*, is to cause a computer to solve a specified problem. The nature of that problem may vary immensely. It should never be forgotten that *programming is not an end in itself.*
- The task of writing a program to solve a particular problem can be broken down into four basic steps:

  (1) **Specification**: Specify the problem clearly.
  (2) **Anaysis and design**: Analyse the problem and break it down into its fundamental elements.
  (3) **Coding**: Code the program according to the plan developed at step 2.
  (4) **Testing**: Test the program exhaustively, and repeat steps 2 and 3 as necessary until the program works correctly in all situations that you can envisage.

- A well-designed program is easier to test, to maintain and to port to other computer systems.
- A **structure plan** is a method for assisting in the design of a program. It involves creating a structure plan of successive levels of refinement until a point is reached

where the programmer can readily code the individual steps without the need for further analysis. This *top-down* approach is universally recognized as being the ideal model for developing programs although there are situations when it is necessary to also look at the problem from the other direction (*bottom-up*). The programming of sub-problems identified during top-down design can be deferred by specifying a subprogram for the purpose.

- The program written is **Example 2.1** is a very simple one, but it does contain many of the basic building blocks and concepts which apply to all Fortran 90 programs. We shall therefore examine it carefully to establish these concepts before we move on to look at the language itself in any detail.
- A program is composed of the **main program unit** and program units of other types, in particular **subroutines**.
- The structure of a main program unit is:

  PROGRAM *name*
  Specification statements
  ...
  Executable statements
  ...
  END PROGRAM *name*

- Every main program unit must start with a PROGRAM statement, and end with an END PROGRAM statement.
- **Specification statements** provide information about the program to the compiler.
- An IMPLICIT NONE statement is a special specification statement which should always immediately follow a PROGRAM statement. It is used to inhibit a particularly undesirable feature of Fortran which is carried over from earlier versions of Fortran.
- A **variable declaration** is a particular specification statement which specifies the data type and name of the variables which will be used to hold (numeric or other) information.
- **Executable statements** are obeyed by the computer during the execution of the program.
- A **list-directed input statement** is a particular executable statement which is used to obtain information from the user of a program during execution through the **default input device** (often the keyboard).
- A **list-directed output statement** is a particular executable statement which is used to give information to the user of a program during execution through the **default output device** (often the screen).
- A CALL statement is used to transfer processing to a **subroutine**, using information passed to the subroutine by means of **arguments**, enclosed in parentheses.
- A Fortran 90 **name** must obey the following rules:
  - it must consist of a maximum of 31 characters;
  - it may only contain the 26 upper case letters A–Z, the 26 lower case letters a–z, the ten digits 0–9, and the underscore character _; upper and lower case letters are considered to be identical in this context;

  - it must begin with a letter.

- **Keywords** are Fortran names which have a special meaning in the Fortran language; other names are called **identifiers**. To assist readibility of the example programs we shall use upper case letters for keywords and lower case letters for identifiers.
- **Blank** characters are significant and must be used to separate names, constants or statement labels from other names, constants or statement labels, and from Fortran keywords. The number of blanks used in this context is irrelevant for the compiler.
- A **comment line** is a line whose first non-blank character is an exclamation mark, !. A **trailing comment** is a comment whose initial ! follows the last statement on a line. Comments are ignored by the compiler. Comments should be used liberally to explain anything which is not obvious from the code itself.
- A line may contain a maximum of **132** characters.
- A line may contain more than one statement, in which case a semicolon, ;, separates each pair of successive statements.
- The presence of an ampersand, &, as the last non-blank character of a line is an indication of a **continuation line**, that is, that the statement is continued on the next line. If it occurs in a character context, then the first non-blank character of the next line must also be an ampersand, and the character string continues from immediately after that ampersand.
- A statement may have a maximum of **39** continuation lines.
- Errors in programs are of different types. A **syntactic error** is an error in the syntax, or grammar, of the statement. A **semantic error** is an error in the logic of the program; that is, it does not do what it was intended to do. **Compilation errors** are errors detected during the compilation process. **Execution errors** are errors that occur during the execution of the compiled program. Compilation errors are usually the result of syntactic errors, although some semantic errors may also be detected. Execution errors are always the result of semantic errors in the program.
- Testing programs is a vitally important part of the programming process. Even with apparently simple programs one should always thoroughly test them to ensure that they produce the correct answers from valid data, and react in a predictable and useful manner when presented with invalid data.
- One shoud never forget that computers have no intelligence; they will only do what you tell them to do – no matter how silly that may be – rather than what you intended them to do.
- The action required to run a Fortran program on a particular computer and to identify any specific requirements will be specific to the particular system and compiler being used.

Fortran 90 syntax introduced in Chapter 2	
Initial statement	PROGRAM *name*
End statement	END PROGRAM *name* END PROGRAM END
Implicit type specification statement	IMPLICIT NONE
Variable declaration statement	REAL :: *list of names*
List-directed input and output statements	READ *, *list of names* PRINT *, *list of names and/or values*
Subroutine call	CALL *subroutine_name* (*argument_1, argument_2, ...*)

Example 2.1

Problem (2.1)

Write a program which will ask the user for the x and y coordinates of three points and which will calculate the equation of the circle passing through those three points, namely

$$(x-a)^2 + (y-b)^2 = r^2$$

and then display the coordinates (a, b) of the centre of the circle and its radius, r.

Analysis (2.1)

Structure plan:

1	Read three sets of coordinates $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3)
2	Calculate the equation of the circle using the procedure *calculate_circle*
3	Display the values a, b and r

Solution (2.1)

```
PROGRAM circle
   IMPLICIT NONE
   !
   ! This program calculates the equation of a circle passing
   ! through three points
   !
   ! Variable declarations
   !
   REAL :: x1, y1, x2, y2, x3, y3, a, b, r
   !
   ! Step 1
   !
   PRINT *, "Please type the coordinates of three points"
   PRINT *, "in the order x1, y1, x2, y2, x3, y3"
   READ *, x1, y1, x2, y2, x3, y3     ! Read the three points
   !
   ! Step 2
   !
   CALL calculate_circle(x1, y1, x2, y2, x3, y3, a, b, r)
   !
   ! Step 3
   !
   PRINT *, "The centre of the circle through these points is &
            &(", a, ",", b, ")"
   PRINT *, "Its radius is ", r
   !
END PROGRAM circle
```

Result of running the Solution (2.1)

```
Please type the coordinates of three points
in the order x1, y1, x2, y2, x3, y3
4.71 4.71
6.39 0.63
0.63 0.63
The centre of the circle through these points is ( 3.510, 1.830)
Its radius is  3.120
```

CHAPTER 3. ESSENTIAL DATA HANDLING

OVERVIEW

There are two fundamental types of numbers in both mathematics and programming – namely those which are whole numbers, and those which are not. In Fortran these are known as integers and real numbers, respectively, and the difference between them is of vital importance in all programming languages. A third fundamental data type allows character information to be stored and manipulated.

This chapter discusses these three basic data types, the ways in which they may be used in calculations or other types of expressions, and the facilities contained within Fortran for the input and output of numeric and textual information.

Finally, an important feature of Fortran 90 is its ability to allow programmers to create their own data types, so that they may more readily express problems in their own terms, rather than in an arbitrary set of more basic functions. This is an important new development in Fortran 90, and one which will be developed further in subsequent chapters.

SUMMARY

- An **integer** is always held *exactly* in the computer's memory, and has a (relatively) limited reange (between about -2×10^9 and $+2 \times 10^9$ on a typical 32-bit computer)
- A **real number** is stored as a floating-point number, is held as an *approximation* to a fixed number of significant digits and has a very large range (typically between about -10^{38} and $+10^{38}$ to seven or eight significant digits on the same 32-bit computer).
- **Variables** are locations in the computer's memory in which variable information may be stored.
- All variables should be declared in a **type declaration statement** before their first use. At its simplest this statement takes the form

  TYPE :: *name*

  or

  TYPE :: *name_1, name_2, ...*

  where *TYPE* specifies the **data type** for which memory space is to be reserved, and *name, name_1, name_2, ...* are the names chosen by the programmer with which to refer to the **variables** that have been declared.
- Example:

```
REAL :: real_1, real_2, real_3
INTEGER :: integer_1, integer_2
```

- An IMPLICIT NONE statement should always be placed immediately after the initial statement of the main program unit to force the compiler to require that all variables appear in a type declaration statement.
- There are only two ways in which a variable can be given a value during the execution of a program – by **assignement** or by a READ statement.

- An **assignement statement** takes the form

  name = *expression*

  where *name* is the name of the variable, and *expression* is an arithmetic expression which will be evaluated by the computer to calculate the value to be assigned to the variable *name*.

- If an integer value is assigned to a real variable it is converted to its real equivalent before assignment; if a real value is assigned to an integer variable it is truncated before conversion to integer, and any fractional part is lost.

- Example:

```
a = b + c*d/e - f**g/h + i*j + k
a = b + (c*d)/e - (f**g)/h + (i*j) + k
```

- Arithmetic operators in Fortran:

Operator	*Meaning*	*Priority*
+	Addition	Low
-	Subtraction	Low
*	Multiplication	Medium
/	Division	Medium
**	Exponentiation	High

- The priority of arithmetic operators in an arithmetic expression is the same as in mathematics, namely exponentiation is carried out first, followed by multiplication and division, followed by addition and subtraction. Within the same level of priority evaluation of the expression will proceed from left to right, except in the case of exponentiation where evaluation proceeds from right to left. The priority may be altered by the use of parentheses.

- If one of the operands of an arithmetic operator is real, then the evaluation of that operation is carried out using real arithmetic, with any integer operand being converted to real.

- The evaluation of a **mixed-mode expression**, where not all the operands are of the same type, proceeds as already defined until a sub-expression is to be evaluated which has two operands of different types. At this point, and not before, the integer value is converted to real.

- The result of the division of two integers (**integer division**) is the integer which is the truncated value of the mathematical value of the division.

- Example:

```
REAL :: temp_C, temp_F, temp_F_1, temp_F_2, temp_F_3, temp_F_4
...
temp_F = 9.0 * temp_C/5.0 + 32.0
temp_F_1 = 9.0/5.0 * temp_C + 32.0   ! temp_F_1 = temp_F
temp_F_2 = 1.8 * temp_C + 32.0       ! temp_F_2 = temp_F
temp_F_3 = 9 * temp_C/5 + 32         ! temp_F_3 = temp_F
temp_F_4 = 9/5 * temp_C + 32         !!! temp_F_4 /= temp_F
```

- All five arithmetic operators are **binary operators**, that is, they have two operands. Addition and subtraction can also be used as **unary operators**, having only one operand.
- Example:

```
p = -q          ! p = 0.0 - q
x = + y         ! x = 0.0 + y
```

- **Constants** are locations in which information is stored which cannot be altered during the execution of the program.
- Constants may have names like variables or they may simply appear in a Fortran statement by writing their value. In this latter case they are called **literal constants** because every digit of the numbers is specified *literally*.
- Numerical literal constants are written in the normal way, and the presence or absence of a decimal point defines the type of constant.
- There is one exception to the rule that real constants must have a decimal point, namely the **exponential form**. This takes the form

  m E e

  where m is called the **mantissa** and e is the **exponent**. The mantissa may be written either with or without a decimal point, whereas the exponent must take the form of an integer.
- Example: 0.000001 can be written

  0.1E-5 or 1.0E-6 or 1E-6 or 100E-8, etc.
- **List-directed input/output statements** have an almost identical syntax:

  READ *, *var_1, var_2, ...*
  PRINT *, *item_1, item_2, ...*
- The main difference between them is that the list of items in a READ statement may only contain variable names, whereas the list in a PRINT statement may also contain constants or expressions. These lists of names and/or other items are referred to as an **input list** and an **output list**, respectively. The asterisk following the READ or PRINT indicates that **list-directed formatting** is to take place. We shall see in Chapter 8 how other forms of input and output formatting may be defined.
- The list-directed READ statement will take its input from a processor-defined input unit known as the **default input unit**, while the list-directed PRINT statement will send its output to a processor-defined unit known as the **default output unit**. In most systems, such as workstations or personal computers, these default units will be the keyboard and display, respectively; we shall see in Chapter 8 how to specify other input or output units where necessary.
- The term 'list-directed' is thus used because the interpretation of the data input, or the representation of the data output, is determined by the list of items in the input or output statement.
- A value that is input to a real variable may contain a decimal point, or the decimal point may be omitted, in which case it is treated as though the integer value read were followed by a decimal point. A value that is to be input to an integer variable must not contain a decimal point, and the occurrence of one will cause an error.

- One important point that must be considered with list-directed input concerns the **termination** of each data value being input. The rule is that each number, or other item, must be followed by a **value separator** consisting of

  a comma, a space, a slash (/) or the end of the line;

  any of these value separators may be preceded or followed by any number of consecutive blanks (or spaces).

- If there are two consecutive commas, then the effect is to read a **null value**, which results in the value of the corresponding variable in the input list being left unchanged. Note that a common cause of error is to believe that the value will be set to zero.

- If the terminating character is a slash then no more data items are read, and processing of the input statement is ended. If there are any remaining items in the input list then the result is as though null values have been input to them; in other words, their values remain unchanged.

- On output, list-directed formatting causes the processor to use an appropriate format for the values being printed. Exactly what form this takes is processor-dependent, but it is usually perfectly adequate for simple programs and for initial testing.

- **Character literal constants** can be used in output statements to provide textual information. They consist of a string of characters chosen from those available to the user on the computer system being used, enclosed between **quotation marks** or **apostrophes**. As long as the same character is used at the beginning and the end it does not matter which is used.

- A single real or integer number is stored in a **numeric storage unit**, which consists of a contiguous area of memory capable of storing 12, 32, 48 or 64 bits, or binary digits. Each character is stored in a **character storage unit**, typically occupying 8 or 16 bits.

- A **character variable** consists of a sequence of one or more consecutive character storage units.

- Programs in the Fortran language are written using characters from the **Fortran Character Set**, constituted by the following 58 characters:

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  (a b c d e f g h i j k l m n o p q r s t u v w x y z)
  0 1 2 3 4 5 6 7 8 9
  _ # = + - * / ( ) , . ' : ! " % & ; < > ? $

  (where # represents the space, or blank, character)

- Most processors also support a number of other characters as part of their **default character set**.

- A character variable is declared in a very similar manner to that used for integer and real numbers, with the important difference that it is necessary to specify how many characters the variable is to be capable of storing:

  CHARACTER(LEN =*length*) :: *name_1, name_2, ...*

This declares one or more CHARACTER variables, each of which has a **length** of *length*.

- The length specification may be either a positive integer constant or an integer constant expression.
- If no length specification is provided, then the length is taken to be one.
- There are two additional, shorter, ways of writing this statement (but the full form is recommended for greater clarity):

  CHARACTER(*length*) :: *name_1, name_2, ...*
  CHARACTER***length* :: *name_1, name_2, ...*

- When assigning a character string to a character variable whose length is not the same as that of the string, the string stored in the variable is extended to the right with blanks, or truncated from the right, so as to exactly fill the character variable to which it is being assigned.
- The form of any character data to be read by a list-directed READ statement is normally the same as that of a character constant. In other words it must be delimited by either quotation marks or by apostrophes. These delimiting characters are not required if *all* of the following conditions are met:
  - the character data is all contained within a single record or line;
  - the character data does not contain any blanks, any commas or any slashes;
  - the first non-blank character is not a quotation mark or an apostrophe;
  - the leading characters are not numeric followed by an asterisk.

  In this case the character constant is terminated by any of the value separators which will terminate a numeric data item (blank, comma, slash or end of record).
- If the character data which is read by a list-directed READ statement is too long or too short for the variable concerned then it is truncated or extended on the right in exactly the same way as for assignement.
- The output situation is rather simpler, and a list-directed PRINT statement will output exactly what is stored in a character variable or constant, including any trailing blanks, without any delimiting apostrophes or quotation marks.
- Two character strings can be combined to form a third, composite string. This process is called **concatenation** and is carried out by means of the **concatenation operator**, consisting of two consecutive slashes.
- **Character substrings** can be identified by following the character variable name or character constant by two integer expressions separated by a colon and enclosed in parentheses. The two integer values represent the positions in the character variable or constant of the first and last characters of the substring. Either may be omitted, but not both, in which case the first or last character position is assumed, as appropriate.

  substring = *string*(*first_position* : *last_position*)
  substring = *string*(*first_position* :)
  substring = *string*(: *last_position*)

- Character substrings may be used wherever the character variables or character constants of which they are substrings may be used.
- A variable declaration may include the specification of an **initial value**.

  type :: *name* = *initial_value*

  Any initial value specified must either be a literal constant or a constant expression, that is an expression whose components are all constants.
- Example:

```
REAL :: a = 0.0,  b,  c = 1.0E-6
INTEGER :: max = 100
CHARACTER(LEN=10) :: name="Undefined"
```

- A **named constant** declaration takes the same form as a variable declaration specifying an initial value, except that the name has the PARAMETER attribute.

  type, PARAMETER :: *name* = *initial_value*
- Example:

```
REAL, PARAMETER :: pi = 3.1415926; pi_by_2 = pi/2.0
INTEGER, PARAMETER :: max_iter = 100
```

- There are six **intrinsic data types** that can be processed by Fortran programs, of which we have met the three major ones: REAL, INTEGER and CHARACTER.
- A **derived type** is a user-defined data type, each of whose components is either an intrinsic type or a previously defined derived type. A derived type is defined by a special sequence of statements, which in their simplest form are as follows:

  TYPE *new_type*
  component_definition
  ...
  END TYPE *new_type*

  There may be as many component definitions as required, and each takes the same form as a variable declaration.
- Example:

```
TYPE person
   CHARACTER :: first_name*12, middle_initial*1, last_name*12
   INTEGER :: age
   CHARACTER :: sex          ! M or F
   CHARACTER(LEN=11) :: social_security
END TYPE person
```

- Variables of a derived type are declared in a similar way to that used for intrinsic types, except that the type name is enclosed in parentheses and preceded by the keyword TYPE:
- Example:

```
TYPE(person) :: jack, jill
```

- Derived type literal constants are specified by means of **structure constructors**: a sequence of constants corresponding to the components of the derived type, enclosed in parentheses and preceded by the type name.
- Example:

```
jack = person("Jack", "R", "Hagenbach", 47, "M", "123-45-6789")
jill = person("Jill", "M", "Smith", 39, "F", "987-65-4321")
```

- We may refer directly to a component of a derived type variable by following the variable by a percentage sign, %, and the name of the component.
- Example:

```
jill%last_name = jack%last_name
```

Fortran syntax introduced in Chapter 3

Derived type definition	TYPE *type_name* *1st_component_declaration* *2nd_component_declaration* ⋮ END TYPE *type_name*
Variable declaration	REAL :: *list of variable names* INTEGER :: *list of variable names* CHARACTER (LEN=length) :: *list of variable names* TYPE(*derived type_name*) :: *list of variable names*
Initial value specification	*type* :: *name* = *initial_value, ...*
Named constant declaration	*type*, PARAMETER :: *name* = *initial_value, ...*
Assignment statement	*variable_name* = *expression*
Character substring specification	*name*(*first_position* : *last_position*) *name*(*first_position* :) *name*(: *last_position*)
Arithmetic operators	**, *, /, +, -
Character operator	//

Example 3.2

Problem (3.2)

Write a program which asks the user for his(her) title, first name and last name, and prints a welcome message using both the full name and first name.

Analysis (3.2)

Structure plan:

1	Read title, first name and last name
2	Concatenate the resulting strings together, using the intrinsic function TRIM to remove trailing blanks from the title and first name
3	Print a welcome message using the formal address, and another using just the first name

Solution (3.2)

```
PROGRAM welcome
   IMPLICIT NONE
   !
   ! This program manipulates character strings to produce a
   ! properly formatted welcome message
   !
   ! Variable declarations
   CHARACTER(LEN=10) :: title
   CHARACTER(LEN=20) :: first_name, last_name
   CHARACTER(LEN=50) :: full_name
   !
   ! Ask for name, etc
   PRINT *, "Please give your full name in the form requested"
   PRINT *, "Title (Mr./Mrs./Ms./etc.): "
   READ *, title
   PRINT *, "First name: "
   READ *, first_name
   PRINT *, "Last name: "
   READ *, last_name
   !
   ! Create full name
   full_name=TRIM(title)//" "//TRIM(first_name)//" "//last_name
   !
   ! Print messages
   PRINT *, "Welcome  ", full_name
   PRINT *, "May I call you  ",TRIM(first_name),"?"
   !
END PROGRAM welcome
```

Example 3.4

Problem (3.4)

Define a data type which can be used to represent complex numbers, and then use it in a program which reads two complex numbers and calculates and prints their sum and their product.

Analysis (3.4)

Given two complex numbers

$$z_1 = x_1 + iy_1, \qquad z_2 = x_2 + iy_2,$$

the rules for addition and multiplication are the following:

$$\begin{aligned} z_1 + z_2 &= x_1 + x_2 + i\,(y_1 + y_2) \\ z_1 \times z_2 &= x_1 \times x_2 - y_1 \times y_2 + i\,(x_1 \times y_2 + x_2 \times y_1) \end{aligned}$$

Structure plan:

1	Define a data type for complex numbers
2	Read two complex numbers
3	Calculate their sum and their product
4	Print results

Solution (3.4)

```
PROGRAM complex_arithmetic
   IMPLICIT NONE
   !
   ! A program to illustrate the use of a derived type to perform
   ! complex arithmetic
   !
   ! Type definition
   TYPE complex_number
      REAL :: real_part, imag_part
   END TYPE complex_number
   !
   ! Variable definitions
   TYPE(complex_number) :: z1, z2, sum, prod
   !
   ! Read data
   PRINT *, "Please supply two complex numbers"
   PRINT *, "Each complex number should be typed as two numbers,"
   PRINT *, "representing the real and imaginary parts of the number"
   READ *, z1, z2
   !
   ! Calculate sum and product
   sum%real_part = z1%real_part + z2%real_part
   sum%imag_part = z1%imag_part + z2%imag_part
   !
   prod%real_part = z1%real_part * z2%real_part - &
                    z1%imag_part * z2%imag_part
   prod%imag_part = z1%real_part * z2%imag_part + &
                    z1%imag_part * z2%real_part
   !
   ! Print results
   PRINT *, "The sum of the two numbers is (", &
             sum%real_part, ", ", sum%imag_part, ")"
   PRINT *, "The product of the two numbers is (", &
             prod%real_part, ", ", prod%imag_part, ")"
   !
END PROGRAM complex_arithmetic
```

CHAPTER 4. BASIC BUILDING BLOCKS

In all walks of life, the easiest way to solve most problems is to break them down into smaller sub-problems and deal with each of these in turn, further subdividing these sub-problems as necessary.

This chapter introduces the concept of a procedure to assist in the solution of such sub-problems, and shows how Fortran's two types of procedures, functions and subroutines, are used as the primary building blocks in well-designed programs.

A further encapsulation facility, known as a module, is also introduced in this chapter as a means of providing controlled access to global data, and is also shown to be an essential tool in the use of derived (or user-defined) datatypes. Modules are also recommended as a means of packaging groups of related procedures, for ease of manipulation, as a means if providing additional security and to simplify the use of some of the powerful features of Fortran 90 that will be met in subsequent chapters.

SUMMARY

- A **procedure** is a special section of a program which is, in some way, referred to whenever required.
- Procedures fall into two broad categories: **intrinsic procedures**, which are part of the Fortran language; **external procedures**, which are written by the programmer (or by some other person who then allows the programmer to use them).
- Procedures are further categorized according to their mode of use into **subroutines** and **functions**.
- There are 108 intrinsic functions and 5 intrinsic subroutines available in Fortran 90.
- The purpose of a function is to take one or more values (or **arguments**) and create a single result (the **function value**).
- A function reference takes the general form:

  name (*argument*)
  name (*arg_1, arg_2, ...*)

- Examples:

  SQRT(x), intrinsic function which calculates the square root of a positive number x

  cube_root(x), external function which calculates the cubic root of a real number x

- A function is used simply by referring to it in an expression in place of a variable or constant.
- Example:

```
- b + SQRT(b*b - 4.0*a*c)
```

- Many intrinsic functions exist in several versions, each of which operates on arguments of different types; such functions are called **generic functions**.

- A Fortran 90 program consists of **one main program unit**, and any number of four other types of program units:
  - **external function subprogram units**,
  - **external subroutine subprogram units**,
  - **module program units**,
  - **block data program units**.
- All the program units have the same broad structure, consisting of an **initial statement**, any **specification statements**, any **executable statements**, and an END statement.
- One of the most important concepts of Fortran is that *one program unit need never be aware of the internal details of any other program unit.* The only link between one program unit and a subsidiary program unit is through the **interface** of the subsidiary program unit, which consists of the name of the program unit and certain other **public** entities of the program unit. This very important principle means that it is possible to write subprograms totally independently of the main program, and of each other. This feature opens up the way for **libraries** of subprograms: collections of subprograms that can be used by more than one program. It also permits large projects to use more than one programmer; all the programmers need to communicate to each other is the information about the interfaces of their procedures.
- The structure of an **external function subprogram** is:

```
type FUNCTION name( dum_1, dum_2, ... )
    IMPLICIT NONE
    ...
    Specification statements, etc.
    ...
    Executable statements
    ...
END FUNCTION name
```

  or

```
FUNCTION name( dum_1, dum_2, ... )
    IMPLICIT NONE
    type :: name
    ...
    Specification statements, etc.
    ...
    Executable statements
    ...
END FUNCTION name
```

  where *dum_1, dum_2, ...* are **dummy arguments** which represent the **actual arguments** which will be used when the function is used (or **referenced**) and *type* is the type of the result of the function.
- The **result variable** is the means by which a function returns its value. Every function *must* contain a variable having the same name as the function, and this variable *must* be assigned, or otherwise given, a value to return as the value of the

function before an exit is made from the function. The type of this result variable may be specified either in the initial FUNCTION statement or in a conventional declaration statement.

- The function *name* must be declared in the calling program unit in a conventional declaration statement in order that the Fortran processor is aware of its type.
- Although it is not necessary, it is possible to add an EXTERNAL attribute specification to such a declaration; this addition informs the compiler that the name is that of a function and not of a variable.

  REAL, EXTERNAL :: *function_name*
- The difference between a function and a subroutine lies in how they are referenced and how the results, if any, are returned.
- A subroutine's arguments are used both to receive information to operate on and to return results.
- A subroutine is accessed by means of a CALL statement, which gives the name of the subroutine and a list of arguments which will be used to transmit information between the calling program unit and the subroutine:

  CALL *name* (*arg_1, arg_2, . . .*)
- A subroutine may have no arguments, in which case the CALL statement takes the form:

  CALL *name*

  or

  CALL *name*()
- A subroutine need not return anything.
- The structure of an **external subroutine subprogram** is:

  SUBROUTINE *name(dum_1, dum_2, . . .)*
  IMPLICIT NONE ...
  Specification statements, etc.
  ...
  Executable statements
  ...
  END SUBROUTINE *name*
- Execution of a program will start at the beginning of the main program unit.
- A function reference and the CALL of a subroutine causes a **transfer of control** so that instead of continuing to process the current statement, the computer executes the statements contained within the function or the subroutine. When the function or the subroutine has completed its task it returns to the calling program unit and execution continues with the next statement.
- Only the arguments of a procedure are accessible outside the procedure.
- A **local variable** or **internal variable** of a procedure in which it is declared has no existence outside the procedure, that is, it is not accessible from outside the procedure.

- Procedures may be referenced in the main program or in another procedure. However a procedure may not refer to itself, either directly or indirectly (for example, through referencing another procedure which, in turn, references the original procedure). This is known as **recursion** and is not allowed unless we take special action to permit it.
- When a function or subroutine is referenced, information is passed to it through its arguments; in the case of a subroutine, information may also be returned to the calling program unit through its arguments. The relationship between the **actual arguments** in the calling program unit and the dummy arguments in the subroutine or function is of vital importance in this process. It is important to realize that the dummy arguments do not exist as independent entities – they are simply a means by which the procedure can identify the actual arguments in the calling program unit. One very important point to stress is that *the order and types of the actual arguments must correspond exactly with the order and types of the corresponding dummy arguments.*
- The INTENT attribute is one of a number of attributes that may follow the *type* in a declaration statement. It may only be used in the declaration of a dummy argument. It is used to control the direction in which the arguments are used to pass information. It can take one of the following three forms:

  INTENT(IN) which informs the processor that this dummy argument is used only to provide information to the procedure, and the procedure will not be allowed to alter its value in any way.

  INTENT(OUT) which informs the processor that this dummy argument will only be used to return information from the procedure to the calling program. Its value will be undefined on entry to the procedure and it must be given a value by some means before being used in an expression, or being otherwise referred to in a context which will require its value to be evaluated.

  INTENT(INOUT) which informs the processor that this dummy argument may be used for transmission of information in both directions.

- A subroutine's arguments may have all three forms of INTENT attribute. The arguments of a function should always be declared with INTENT(IN).
- Arguments of procedures may also be of character data type. In this case it is convenient to use an **assumed-length character declaration** in the procedure, that is, the character string assumes its length from the corresponding actual argument when the procedure is executed.

```
CHARACTER(LEN = *) :: character_dummy_argument
```

- One of the great advantages of subprograms is that they enable us to break the design of a program into several smaller, more manageable sections, and then to write and test each of these sections independently of the rest of the program. This paves the way for an approach known as **modular program development**, which is a key concept of software engineering. This approach breaks the problem down into its major sub-problems, or **components**, each of which can then be dealt with independently of the others.

- As a rule of thumb, we would suggest that no procedure should be longer than about 50 lines, excluding any comments, so that it can be printed on a single sheet of paper or viewed easily on a screen.
- A MODULE is another form of program unit which is used for rather different purposes than a procedure. One very important use of modules relates to **global accessibility** of variables, constants and derived type definitions: by using a module one can make some or all the entities declared within it accessible to more than one program unit. Access is by means of an appropriate USE statement:

  USE *name*

  where *name* is the name of the module in which the variables, constants, and/or derived data type definitions are declared. Entities which are made available in this way are said to be made available by USE **association**.

  The USE statement comes after the initial statement (PROGRAM, SUBROUTINE or FUNCTION) but before any other statements.
- The broad structure of a module is:

  ```
  MODULE name
      IMPLICIT NONE
      SAVE
      ...
      Other specification statements, etc.
      ...
      Executable statements
      ...
  END MODULE name
  ```
- The statement consisting of the single word SAVE should always be included in any module which declares any variables.
- One module can USE another module in order to gain access to items declared within it, and those items then also become available along with the modules own entities.
- A module may not USE itself, either directly or indirectly (via a recursive chain of other modules).
- Objects of derived types can only be used as arguments to procedures if their type is defined in a MODULE which is used by the relevant program units.
- It is desirable for some security aspects, and essential for some of the language features that will be met in future chapters, that procedures have an **explicit interface**. One way that we can always make the interface of a procedure explicit is by placing the procedure in a module. The rules relating to modules specify that
  - the interfaces of all the procedures defined within a single module are explicit to each other;
  - the interfaces of any procedures made available by USE association are explicit in the program unit that is using the module.

- The statement consisting of the single word

  CONTAINS

  should be placed before the first procedure in a module that contains procedures.
- Modules are of great assistance in the design and control of data as they enable a programmer to group the data in such a way that all those procedures that require access to a particular group can do so by simply using the appropriate module.

Fortran 90 syntax introduced in Chapter 4

Initial statements	SUBROUTINE *name*(*dummy argument list*) SUBROUTINE *name* *type* FUNCTION *name*(*dummy argument list*) *type* FUNCTION *name*() FUNCTION *name*(*dummy argument list*) FUNCTION *name*() MODULE *name*
Function reference	*function_name*(*actual argument list*) *function_name*()
Subroutine call	CALL *subroutine_name*(*actual argument list*) CALL *subroutine_name*()
Module use	USE *module_name*
Assumed length character declaration	CHARACTER(LEN = *) :: *character_dummy_arg* CHARACTER * (*) :: *character_dummy_arg*
Argument intent attribute	INTENT(*(intent*) where *intent* is IN, OUT or INOUT
External procedure attribute	EXTERNAL
SAVE statement	SAVE
CONTAINS statement	CONTAINS

Example 4.1x

Problem (4.1x)

Write a program which will demonstrate the use of the function *cube_root* to calculate the cube root of a positive real number.

Analysis (4.1x)

Structure plan:

1	Read positive real number *pos_num*.
2	Obtain *root_3* by reference to the function *cube_root*.
3	Print *pos_num* and *root_3*.

Example 4.2x

Problem (4.2x)

Write a program which will demonstrate the use of the subroutine *roots* to calculate the square root, the cube root and the fourth root of a positive real number.

Analysis (4.2x)

Structure plan:

1	Read positive real number *pos_num*.
2	Obtain *root_2*, *root_3* and *root_4* by calling the subroutine *roots*.
3	Print *pos_num*, *root_2*, *root_3* and *root_4*.

Example 4.1x

```
PROGRAM function_demo
   IMPLICIT NONE
   !
   ! A program to demonstrate the use of the function cube_root
   !
   ! Variable declarations
   REAL, EXTERNAL :: cube_root
   REAL :: pos_num, root_3
   !
   ! Get positive number from user
   PRINT *, "Please type a positive real number: "
   READ *, pos_num
   !
   ! Obtain root
   root_3=cube_root(pos_num)
   !
   ! Display number and its root
   PRINT *, "The cube root of ", pos_num, " is ", root_3
   !
END PROGRAM function_demo


REAL FUNCTION cube_root(x)            !!! FUNCTION cube_root(x)
   IMPLICIT NONE
   !
   ! Function to calculate the cube root of a positive real number
   !
   !                                  !!!   REAL :: cube_root
   !
   ! Dummy argument declaration
   REAL, INTENT(IN) :: x
   !
   ! Local variable declaration
   REAL :: log_x
   !
   ! Calculate cube root by using logs
   log_x = LOG(x)
   cube_root = EXP(log_x/3.0)
   !
END FUNCTION cube_root
```

Example 4.2x

```
PROGRAM subroutine_demo
   IMPLICIT NONE
   !
   ! A program to demonstrate the use of the subroutine roots
   !
   ! Variable declarations
   REAL :: pos_num, root_2, root_3, root_4
   !
   ! Get positive number from user
   PRINT *, "Please type a positive real number: "
   READ *, pos_num
   !
   ! Obtain roots
   CALL roots(pos_num, root_2, root_3, root_4)
   !
   ! Display number and its roots
   PRINT *, "The square root of ", pos_num, " is ", root_2
   PRINT *, "The cube root of ",   pos_num, " is ", root_3
   PRINT *, "The fourth root of ", pos_num, " is ", root_4
   !
END PROGRAM subroutine_demo

SUBROUTINE roots(x, square_root, cube_root, fourth_root)
   IMPLICIT NONE
   !
   ! Subroutine to calculate various roots of a positive real number,
   ! supplied as the first argument, and return them in the
   ! second to fourth arguments
   !
   ! Dummy argument declarations
   REAL, INTENT(IN) :: x
   REAL, INTENT(OUT) :: square_root, cube_root, fourth_root
   !
   ! Local variable declarations
   REAL :: log_x
   !
   ! Calculate square root using intrinsic SQRT
   square_root = SQRT(x)
   !
   ! Calculate other roots by using logs
   log_x = LOG(x)
   cube_root = EXP(log_x/3.0)
   fourth_root = EXP(log_x/4.0)
   !
END SUBROUTINE roots
```

Example 4.4

Problem (4.4)

Write two functions for use in a complex arithmetic package using the complex_number derived type which was created in Example 3.4. The functions should each take two complex arguments and return as their result the result of adding and multiplying the two numbers.

Analysis (4.4)

This was already done in Example 3.4.

Structure plan:

1	Place the derived type complex_number in a MODULE complex_data for USE association by the program and the functions
2	Read two complex numbers
3	Calculate their sum using FUNCTION c_add
4	Calculate their product using FUNCTION c_mult
5	Print the results

Solution (4.4)

```
MODULE complex_data
   IMPLICIT NONE
   SAVE
   !
   TYPE complex_number
   REAL :: real_part, imag_part
   END TYPE complex_number
   !
END MODULE complex_data


PROGRAM complex_example
   USE complex_data
   IMPLICIT NONE
   !
   TYPE(complex_number), EXTERNAL :: c_add, c_mult
   TYPE(complex_number) :: z1, z2
   !
   PRINT *, "Please supply two complex numbers as two pairs &
            &of real numbers"
   PRINT *, "Each pair represents the real and imaginary parts &
            &of a complex number"
   READ *, z1, z2
   !
   ! Calculate and print sum and product
   PRINT *, "The sum of the two numbers is ", c_add(z1, z2)
   PRINT *, "The product of the two numbers is ", c_mult(z1, z2)
   !
END PROGRAM complex_example
```

```
FUNCTION c_add(z1, z2)
   USE complex_data
   IMPLICIT NONE
   !
   TYPE(complex_number) :: c_add
   TYPE(complex_number), INTENT(IN) :: z1, z2
   !
   c_add%real_part = z1%real_part + z2%real_part
   c_add%imag_part = z1%imag_part + z2%imag_part
   !
END FUNCTION c_add


FUNCTION c_mult(z1, z2)
   USE complex_data
   IMPLICIT NONE
   !
   TYPE(complex_number) :: c_mult
   TYPE(complex_number), INTENT(IN) :: z1, z2
   !
   c_mult%real_part = z1%real_part * z2%real_part - &
                      z1%imag_part * z2%imag_part
   c_mult%imag_part = z1%real_part * z2%imag_part + &
                      z1%imag_part * z2%real_part
   !
END FUNCTION c_mult
```

CHAPTER 5. CONTROLLING THE FLOW OF YOUR PROGRAM

OVERVIEW

Up to now, our programs have started at the beginning and proceeded to the end without interruption. However, in practice, most problems require us to choose between alternative courses of action, depending upon circumstances which are not determined until the program is executed. The ability of a program to specify how these decisions are to be made is one of the most important aspects of programming.

This chapter introduces the concept of comparison between two numbers or two character strings, and explains how such comparisons can be used to determine which one of two, or more, alternative sections of code are obeyed.

An alternative form of choice, which was not available in earlier versions of Fortran, uses a list of possible values of some variable or expression to determine which of the several alternative blocks of code is actually executed.

SUMMARY

- The ability of a computer program to choose which one of two or more alternative sequences of statements to obey is a major factor in making computers such powerful tools.
- Fortran 90 has a very smiliar construction to that used in the English language to provide alternatives:

```
IF (criterion_1) THEN
      action_1
ELSE IF (criterion_2) THEN
      action_2
ELSE IF (criterion_3) THEN
      action_3
ELSE
      action_4
END IF
```

- The criteria on which the decisions will be based consist of a new type of expression – **a logical expression**. A logical expression can take one of the **logical values**, *true* or *false*.
- A **relational expression** is the simplest form of logical expression in which **relational operators** are used to derive logical values from a comparison of two numeric expressions.

a < b	is	*true*	if a is less than b
a <= b	is	*true*	if a is less than or equal to b
a > b	is	*true*	if a is greater than b
a >= b	is	*true*	if a is greater than or equal b
a == b	is	*true*	if a is equal b
a / = b	is	*true*	if a is not equal to b

- All arithmetic operations have a higher priority than any relational operators and the arithmetic expression, or expressions, are therefore evaluated before any comparisons take place.
- Example: The following relational expressions are identical in their effects:

```
b**2  >=  4*a*c
b**2 - 4*a*c  >= 0
```

- A relational operator may also be used to compare two character expressions by using: (i) the Fortran collating sequence; (ii) the ASCII collating sequence, with the help of special intrinsic functions. (. . .)
- **Logical variables** are declared as

  LOGICAL :: *var_1, var_2, ...*

- Logical variables take one of two values: .TRUE. or .FALSE.
- We are also allowed to write functions which deliver a logical value:

  LOGICAL FUNCTION *logical_fun*(*arg_1, arg_2, . . .*)
  ...

  or

  FUNCTION *logical_fun*(*arg_1, arg_2, . . .*)
  LOGICAL :: *logical_fun*
  ...

- **Logical operators**, .OR., .AND., .EQV., .NEQV., are used to combine two logical expressions or values, and thus to allow more complex comparisons:

L1	L2	L1.OR.L2	L1.AND.L2	L1.EQV.L2	L1.NEQV.L2
true	true	true	true	true	false
true	false	true	false	false	true
false	true	true	false	false	true
false	false	false	false	true	false

- The logical operator .NOT. is a unary operator, and has a single operand: it inverts the value of the following logical expression.
- The logical operators table of priorities is the following:

Operator	*Priority*
.NOT.	highest
.AND.	medium
.OR.	medium
.EQV. and .NEQV.	lowest

- Any arithmetic operators or relational operators (*in that order*) have a higher priority than any logical operators.
- Example:

```
(a < b) .OR. (c < d)
(x <= y) .AND. (y <= z)
```

- The **block IF construct** has the following structure:

  IF (*logical_expression*) THEN
  block of Fortran statements
  ELSE IF (*logical_expression*) THEN
  block of Fortran statements
  ...
  ELSE
  block of Fortran statements
  END IF

  - There may be any number of ELSE IF statements, each followed by a block of statements, or there may be none.
  - There may be one ELSE statement, followed by a block of statements, or there may be none. If there is one, then it, and its succeeding block of statements, must follow all ELSE IF blocks.
  - The block of statements following the IF statement will be executed if the associated logical expression is true.
  - The block of statements following an ELSE IF statement will be executed if the associated logical expression is true, and if the logical expressions in the initial IF statement and in any preceding ELSE IF statements are false.
  - The block of statements following the ELSE statement will be executed only if the logical expressions in all preceeding IF and ELSE IF statements are false.

- The **Logical IF statement** has the structure:

  IF (*logical_expression) Fortran statement*

  It is exactly equivalent (a "shorthand version") to a minimal block IF construct with a single statement block:

  IF (*logical_expression*) THEN
  Fortran statement
  END IF

  Because it is more compact, it can be used in a number of situations without any loss of clarity or efficiency. There are restrictions on the Fortran statement that can be used.

- The CASE construct has the following structure:

  SELECT CASE (*case_expression*)
  CASE (*case_selector*)
  block of Fortran statements
  ...
  CASE DEFAULT
  block of Fortran statements
  END SELECT

  - *case_expression* is either an integer expression, a character expression or a logical expression; real expressions are *prohibited* for this purpose.

- The *case_selector* determines which, if any, of the blocks of statements will be obeyed. The *case_selector* can take one of four forms:

  case_value
  low_value :
  : high_value
  low_value : *high_value*

  or it may be a list of any combination of these. Only the first form is permitted for logical values (since it would be meaningless to list more than one of the possible two values). The meaning of these four alternatives is as follows:

  (1) If the *case_selector* takes the form *case_value* then the following block of code is executed if and only if *case_expression* = *case_value*, where *case_expression* is an integer expression or a character expression, and if and only if *case_expression* .EQV. *case_value*, where it is a logical expression.

  (2) If the *case_selector* takes the form *low_value*: then the following block of code is executed if and only if *low_value* <= *case_expression*.

  (3) If the *case_selector* takes the form :*high_value* then the following block of code is executed if and only if *case_expression* <= *high_value*.

  (4) If the *case_selector* takes the form *low_value*:*high_value* then the following block of code is executed if and only if *low_value* <= *case_expression* .AND. *case_expression* <= *high_value*.

- If none of the specified values or value ranges matches the value of the *case_expression* then the block of code following the CASE DEFAULT statement, if any, is executed; if there is no CASE DEFAULT statement then an exit is made from the CASE construct without any code being executed.
- The order in which the various CASE statements, and their following blocks of statements are written, does not matter, since the rules governing CASE constructs require that there is no overlap. However, we recommend that, for clarity, any CASE DEFAULT statement be placed either as the first CASE statement, or as the last.

- Both the block IF construct and the CASE construct provide the means for a program to select one from a set of blocks of statements and executing that block, or of executing none of them if none of the decision criteria is satisfied.
- One difference between the two constructs is that in the CASE construct the decision criteria must not overlap.
- The other major difference is that the expression which determines the selection must be a logical expression in a block IF construct, but may be an integer expression, a character expression or a logical expression, in a CASE construct (but not a real expression).

Fortran 90 syntax introduced in Chapter 5

Variable declaration	LOGICAL :: *list of variable names*

Block IF construct	IF (*logical_expression*) THEN
	  block_of_code
	ELSE IF (*logical_expression*) THEN
	  block_of_code
	⋮
	ELSE
	  block_of_code
	END IF

CASE construct	SELECT CASE (*case_expression*)
	CASE (*case_selector*)
	  block_of_code
	⋮
	CASE DEFAULT
	  block_of_code
	END SELECT

Logical IF statement	IF (*logical_expression) Fortran statement*
Relational operators	$>$, $>=$, $<=$, $<$, $==$, $/=$ .GT., .GE., .LE., .LT., .EQ., .NE.
Logical operators	.AND., .OR., .EQV., .NEQV., .NOT.

Example 5.6

Problem (5.6)

Write a program to read the coefficients of a quadratic equation and print the roots.

Analysis (5.6)

The program will use the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where

$$ax^2 + bx + c = 0 \qquad \text{and} \qquad a \neq 0.$$

There are three possible cases:

(1)	$b^2 - 4ac \geq \varepsilon$	Equation has two real roots
(2)	$\lvert b^2 - 4ac \rvert < \varepsilon$	Equation has one real root
(3)	$b^2 - 4ac \leq -\varepsilon$	Equation has no real roots

ε is a very small positive number

Structure plan (solution using a block IF construct):

```
1  Read the three coefficients a, b and c
2  Calculate d = b^2 - 4ac
3      If d ≥ epsilon then
       3.1 Calculate and print two roots
       but if d > -epsilon then
       3.2 Calculate and print a single root
       otherwise
       3.3 Print a message to the effect that there are no roots
```

Structure plan (solution using a CASE construct):

```
1  Read the three coefficients a, b and c
2  Calculate d = b^2 - 4ac
3  Calculate selector (d/epsilon)
4  Select case on selector
     4.1 selector > 0
         Calculate and print two roots
     4.2 selector = 0
         Calculate and print a single root
     4.3 selector < 0
         Print a message to the effect that there are no roots
```

Solution (5.6a)

```
PROGRAM quadratic_by_block_IF
   IMPLICIT NONE
   !
   ! A program to solve a quadratic equation using a block IF
   ! statement to distinguish between the three cases
   !
   ! Constant declarations
   REAL, PARAMETER :: epsilon = 1.0E-6
   !
   ! Variable declarations
   REAL :: a, b, c, d, sqrt_d, x1, x2
   !
   ! Read coefficients
   PRINT *, "Please type the three coefficients a, b and c"
   READ *, a, b, c
   !
   ! Calculate b**2-4*a*c
   d = b**2 - 4.0 * a * c
   !
   ! Calculate and print roots, if any
   IF (d >= epsilon) THEN
      ! Two roots
      sqrt_d = SQRT(d)
      x1 = (-b + sqrt_d)/(a+a)
      x2 = (-b - sqrt_d)/(a+a)
      PRINT *, "The equation has two roots: ", x1, " and ", x2
   ELSE IF (d > -epsilon) THEN
      ! One root
      x1 = -b/(a+a)
      PRINT *, "The equation has one root: ", x1
   ELSE
      ! No roots
      PRINT *, "The equation has no real roots"
   END IF
   !
END PROGRAM quadratic_by_block_IF
```

Solution (5.6b)

```
PROGRAM quadratic_by_case
   IMPLICIT NONE
   !
   ! A program to solve a quadratic equation using a CASE
   ! statement to distinguish between the three cases
   !
   ! Constant declarations
   REAL, PARAMETER :: epsilon = 1.0E-6
   !
   ! Variable declarations
   REAL :: a, b, c, d, sqrt_d, x1, x2
   INTEGER :: selector
   !
   ! Read coefficients
   PRINT *, "Please type the three coefficients a, b and c"
   READ *, a, b, c
   !
   ! Calculate b**2-4*a*c and resulting case selector
   d = b**2 - 4.0 * a * c
   selector = d/epsilon
   !
   ! Calculate and print roots, if any
   SELECT CASE (selector)
   CASE (1:)
      ! Two roots
      sqrt_d = SQRT(d)
      x1 = (-b + sqrt_d)/(a+a)
      x2 = (-b - sqrt_d)/(a+a)
      PRINT *, "The equation has two roots: ", x1, " and ", x2
   CASE (0)
      ! One root
      x1 = -b/(a+a)
      PRINT *, "The equation has one root: ", x1
   CASE (:-1)
      ! No roots
      PRINT *, "The equation has no real roots"
   END SELECT
   !
END PROGRAM quadratic_by_case
```

CHAPTER 6. REPEATING PARTS OF YOUR PROGRAM

OVERVIEW

A very large proportion of mathematical techniques rely on some form of iterative process, while the processing of most types of data requires the same, or similar, actions to be carried out repeatedly for each set of data. One of the most important of all programming concepts, therefore, is the ability to repeat some sequences of statements either a predetermined number of times or until some condition is satisfied.

Fortran has a very powerful, yet simple to use, facility for controlling the repetition of blocks of code, and this chapter explains how this facility can be used to control iterative processes as well as more simple repetitive tasks.

The use of repetitive techniques, however, often leads to situations in which it is required to end the repetition earlier than had been anticipated, and Fortran contains a number of statements to assist in these exceptional cases. By their nature, however, such statements interrupt the normal flow of the program and must be used with care if they are not to lead to other problems.

SUMMARY

- A sequence of statements which are repeated is called a **loop**.
- The DO **construct** provides the means for controlling the repetition of statements within a loop. It takes the form:

  DO *count = initial, final, inc*
  ...
  block of statements
  ...
  END DO

- A loop created by use of a DO construct is called a DO **loop**.
- The first statement of a DO loop is called a DO **statement**. It can have the following alternative forms:

  DO *count = initial, final, inc*
  DO *count = initial, final*
  DO

- The first two alternatives define a **count-controlled** DO **loop**, in which an integer variable, *count*, known as the DO **variable**, is used to determine how many times the block of statements which appear between the DO statement and the END DO statement is to be executed. *initial, final* and *inc* must be integer expressions. If *inc* is absent then its value is taken as 1.
- Informally this process means that the loop is executed for *count* taking the value *initial* the first time that the loop is executed, *initial* + *inc* the next time, and so on, with the value of *count* being incremented by *inc* for each subsequent pass; the final pass through the loop will be the one which would result in the *next* pass having a value of *count* greater than *final*.

- The formal definition of this process is that when the DO statement is executed an **iteration count** is first calculated using the formula

  MAX((*final-initial+inc*)/*inc*, 0)

  and the loop is executed that many times. On the first pass the value of *count* is *initial*, and on each subsequent pass its value is increased by *inc*. On normal completion of a count-controlled DO loop the DO variable will have the value that it would have had on the next pass through the loop, had there been one.

- Example:

DO *statement*	*Iteration count*	DO *variable values*
DO i = 1, 10	10	1,2,3,4,5,6,7,8,9,10
DO j = 20, 50, 5	7	20,25,30,35,40,45,50
DO p = 7, 19, 4	4	7,11,15,19
DO q = 4, 5, 6	1	4
DO r = 6, 5, 4	0	6
DO x = -20, 20, 6	7	-20,-14,-8,-2,4,10,16
DO n = 25, 0, -5	6	25,20,15,10,5,0
DO m = 20, -20, -6	7	20,14,8,2,-4,-10,-16

- It is not permitted for the program to alter the value of the DO variable between the initial DO statement and the corresponding END DO statement by any means other than the automatic incrementation which is part of the DO loop processing.

- There are no restrictions on the types of statements that may appear in the block of statements which constitute the **range** of the DO loop. In particular, DO loops may be **nested** within a DO loop, although the whole of the nested loop must, of course, lie within the outer loop.

- In the situations in which it is not possible to determine the number of times that the loop is executed in advance, one can use the third form of the DO statement, together with a new statement EXIT, which causes a **transfer of control** to the statement immediately following the END DO statement.

- Example:

```
DO
   ...
   IF(term < epsilon) EXIT
   ...
END DO
! After obeying the EXIT statement execution continues
! from the next statement
...
```

- This non-counting form of the DO statement should only be used when the programmer can be absolutely certain that there is no possible situation in which the terminating condition will not occur and the loop will become what is known as an **infinite loop**. Since such a 100% certainty is rare, it is recommended that such loops should normally contain a **fail-safe mechanism** in which a DO variable is used to limit the number of repetitions to a predefined maximum.

- Example:

```
DO count=1, max_iterations
   ...
   IF(term < epsilon) EXIT
   ...
END DO
! After obeying the EXIT statement, or after obeying the loop
! max_iteration times, execution continues from the next
! statement
...
```

- The statement CYCLE is similar to the EXIT statement, except that instead of transferring control to the statement *after* the END DO statement it transfers control back to the start of the DO loop in exactly the same way as if it, in fact transferred control to the END DO statement.
- It is possible to give a name to a block DO construct, by preceding the DO statement by a name which is separated from the DO by a collon, and by following the corresponding END DO by the same name:

  block_name: DO
  ...
  END DO *block_name*

- The CYCLE and EXIT statements may also be followed by the name of an enclosing DO construct, in which case control is transferred to, or after, respectively, the END DO statement having the same name.
- A similar naming facility also exists for the block IF and CASE constructs, but in these cases the names are purely for clarity in the case of complex structures.
- The STOP statement causes an immediate termination of the execution of the program.
- The RETURN statement causes an immediate termination of the current procedure.
- A GOTO statement, or GO TO statement, transfers execution to the statement in the same procedure having a specified label.
- A **statement label** may be used to identify a statement. A statement label consists of from one to five consecutive digits, representing a number in the range 1 to 99999, and which precedes the statement being labelled, with at least one space between the two. Every statement label in a program unit must be unique.

Fortran 90 syntax introduced in Chapter 6

Block DO construct	DO *do_var* = *initial, final, inc* ⋮ END DO DO *do_var* = *initial, final* ⋮ END DO DO ⋮ END DO
Loop control statements	EXIT CYCLE
Named block construct statements	*do_block_name*: DO *do_var* = *initial, final, inc* *do_block_name*: DO *do_var* = *initial, final* *do_block_name*: DO EXIT *do_block_name* CYCLE *do_block_name* END DO *do_block_name* *if_block_name*: IF (*logical_expression*) THEN ELSE IF (*logical_expression*) THEN *if_block_name* ELSE *if_block_name* END IF *if_block_name* *case_block_name*: SELECT CASE (*case_expression*) CASE (*case_selector*) *case_block_name* CASE DEFAULT *case_block_name* END SELECT *case_block_name*
STOP statement	STOP
RETURN statement	RETURN
GOTO statement	GOTO *label* GO TO *label*

Example 6.1

Problem (6.1)

Write a program which first reads the number of people sitting an exame. It should then read their marks (or scores) and print the highest and lowest marks, followed by the average mark for the class.

Analysis (6.1)

This is a straightforward problem which will use a DO loop to repeatedly read a mark and use it to update the sum of the marks, the maximum mark so far, and the minimum mark so far.

Structure plan:

1 Initialize *sum* to zero, *maximum* to a large negative value,
minimum to a large positive value
2 Read number of examinees (*number*)
3 Repeat *number* times
  3.1 Read a mark
  3.2 Add it to cumulative sum
  3.3 If it is larger than maximum mark so far set maximum to this mark
  3.4 If it is smaller than minimum mark so far set minimum to this mark
4 Calculate average
5 Print maximum, minimum and average marks

Solution (6.1)

```
PROGRAM examination_marks
   IMPLICIT NONE
   !
   ! This program prints statistics about a set of exam results
   !
   ! Variable declarations
   INTEGER :: i, number, mark, &
              sum = 0, maximum = -HUGE(1), minimum = HUGE(1)
   REAL :: average
   !
   ! Read number of marks, and then the marks
   PRINT *, "How many marks are there? "
   READ *, number
   PRINT *, "Please type ", number, " marks: "
   !
   ! Loop to read and process marks
   DO i = 1, number
      READ *, mark
      !
      SELECT CASE(mark)
      CASE (0:100)
         ! On each pass, update sum, maximum and minimum
         sum = sum + mark
         IF (mark > maximum) maximum = mark
         IF (mark < minimum) minimum = mark
      CASE DEFAULT
         PRINT *, "Invalid mark - Please re-enter data"
         CYCLE
      END SELECT
      !
   END DO
   !
   ! Calculate average mark and output results
   average = REAL(sum)/number
   PRINT *, "Highest mark is ", maximum
   PRINT *, "Lowest mark is ", minimum
   PRINT *, "Average mark is ", average
   !
END PROGRAM examination_marks
```

Example 6.2x

Problem (6.2x)

Write a program to generate the Fibonacci sequence of numbers,

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \text{etc.}$$

until the absolute value of the difference between the ratio of two consecutive numbers $\frac{x_{n+1}}{x_n}$ and the Golden Ratio $\frac{\sqrt{5}+1}{2}$ is less than some arbitrary small value ε.

Analysis (6.2x)

The Fibonacci sequence is generated by the formula

$$x_{n+1} = x_n + x_{n-1}, \qquad n = 1, 2, \ldots$$

The exit criterion will be:

$$\left| \frac{x_{n+1}}{x_n} - \frac{\sqrt{5}+1}{2} \right| < \varepsilon$$

Structure plan:

1	Initialize *x1* and *x2* to 1
2	Read *epsilon* and *max_iterations*
3	Repeat *max_iterations* times:
	3.1 Calculate next term *x =x1+x2* in the sequence
	3.2 Calculate *ratio* and *error*
	3.3 If *error <= epsilon* then exit
	3.4 Otherwise, set *x1=x2* and *x2=x*
4	Print results with indication of ERROR if the loop was executed *max_iterations* times without the exit condition being verified

Solution (6.2x)

```
PROGRAM Fibonacci_sequence
   IMPLICIT NONE
   !
   ! This program illustrates that the ratio of two consecutive
   ! numbers of the Fibonacci sequence converges to the
   ! Golden-Ratio
   !
   ! Constant and variable declarations
   INTEGER :: max_iterations, count, x, x1=1, x2=1
   REAL :: epsilon, ratio, error, golden_ratio
   !
   golden_ratio = (1.0+SQRT(5.0))/2.0
   !
   ! Read epsilon and max_iterations
   PRINT*, "epsilon=?, max_iterations=?"
   READ*,   epsilon, max_iterations
   !
   ! Loop to obtain sequence and error
   DO count=1, max_iterations
      x=x1+x2
      ratio=REAL(x)/REAL(x2)
      error=ABS(ratio-golden_ratio)
      IF(error <= epsilon) EXIT
      x1=x2
      x2=x
   END DO
   !
   ! Output results
   IF(count == max_iterations+1) THEN
      PRINT*, count, x1, x2, x, error, "  ERROR"
   ELSE
      PRINT*, count, x1, x2, x, error
   END IF
   !
END PROGRAM Fibonacci_Sequence
```

CHAPTER 7. AN INTRODUCTION TO ARRAYS

OVERVIEW

In scientific and engineering computing it is commonly necessary to manipulate ordered sets of values, such as vectors and matrices. There is also a common requirement in many applications to repeat the same sequence of operations on successive sets of data.

In order to handle both of these requirements, Fortran provides extensive facilities for grouping a set of items of the same type as an array which can be operated on either as an object in its own right, or by reference to each of its individual elements.

This chapter explains the principles of Fortran 90's array processing features. These are considerably more powerful than those of any other programming language, and include the construction of array-valued constants, the input and output of arrays, the use of arrays as arguments to procedures, and the returning of an array as the result of a function. For ease of comprehension, the description in this chapter is restricted to arrays having one subscript only; arrays having more than one subscript will be discussed in Chapter 13.

SUMMARY

- An **array** is an ordered set of related variables which have the same name and type.
- The individual items within an array are called **array elements**.
- Array elements are identified by following the name of the array by an integer **subscript expression**, enclosed in parentheses.

  array_name(*integer_expression*)

  Function references are allowed as part of the subscript expression, as are array elements (including elements of the same array).
- Example: x, y, z are arrays; i, j, k are integer variables

```
x(10)
y(i+4)
z( 3*i + MAX(i,j,k) )
x( INT( y(i)*z(j)+x(k) ) )
```

- The subscript of an array may take values between a **lower bound** and an **upper bound**.
- The **size** or **extent** of an array is the number of the elements in the array (it is equal to the difference between the upper and lower bounds plus one). These terms will have different meanings for arrays having more than one subscript.
- The **shape** of an array is determined by its extent; it can be stored in an array of size one whose only element is the extent of the array.
- The declaration of an array can be done using a **dimension attribute** or an **array specification** and must specify the bounds for the subscript values.

  type, DIMENSION(*extent*) :: *list of names*
  type, DIMENSION(*lower_bnd* : *upper_bnd*) :: *list of names*
  type :: *name_1*(*extent_1*), *name_2*(*extent_2*), ...
  type :: *name_1*(*l_bnd_1* : *u_bnd_1*), *name_2*(*l_bnd_2* : *u_bnd_2*), ...

- Example:

```
REAL, DIMENSION(50) :: a, b, c
REAL, DIMENSION(11:60) :: d, e, f
                                          ! are equivalent to
REAL :: a(50), b(50), c(50)
REAL :: d(11:60), e(11:60), f(11:60)
```

- An array-valued constant is specified by an **array constructor**, which may include one or more implied DO elements. The list of values in an array constructor must contain exactly the same number of values as the size of the array to which it is being assigned in either an assignement statement or in an initial value assignement in the array declaration statement.

  array_name = (/ *list of values* /)
  array_name = (/ (*value_list, implied_do_control*) /)

  where the *implied_do_control* takes exactly the same structure as the DO variable control specification in a DO statement.

- Example:

```
INTEGER, DIMENSION(10) :: array_1=(/ 1,2,3,4,5,6,7,8,9,10 /)
INTEGER, DIMENSION(10) :: array_1=(/ (i, i=1,10) /)
INTEGER, DIMENSION(50) :: array_2=(/ (0, i=1,50) /)
INTEGER, DIMENSION(100) :: array_3= &
                           (/ ( (0, i=1,9), 10*j, j=1,10 ) /)
```

- Input and output of arrays may be specified element-by-element, by whole arrays, or by use of an implied DO.

  READ *, *array_element, array_name*
  PRINT *, (*array_element_list, implied_do_control*)

- An array element can be used anywhere that a scalar variable can be used.
- Fortran 90 enables an array to be treated as a single object in its own right, in much the same way a scalar object.
- The rules for working with **whole arrays** are:
  - Two arrays are **conformable** if they have the same shape; a scalar, including a constant, is conformable with any array.
  - All intrinsic operations are defined between two conformable objects.
  - Intrinsic operations on arrays take place element-by-element.
- Example:

```
...
REAL :: a(1:20), b(1:20), c(1:20), d(1:20)
...
a = c * d
...
DO i=1,20
   b(i) = c(i) * d(i)                    ! b(i) = a(i)
```

```
   END DO
   ...


   ...
   REAL :: a(1:20), b(0:19), c(10:29), d(-9:10)
   ...
   a = c * d
   ...
   DO i=1,20
      b(i-1) = c(i+9) * d(i-10)            ! b(i) = a(i)
   END DO
...
```

- An **elemental intrinsic procedure** may use an array as an argument in just the same way as it uses a scalar, delivering array-valued results.
- Example:

```
   ...
   array_2=SIN(array_1)
   ...
   DO i=1,size
      array_3(i) = SIN(array_1(i))     ! array_3(i) = array_2(i)
   END DO
   ...
```

- An **explicit-shape array** is an array whose bounds are specified explicitly.
- An **assumed-shape array** is a dummy argument array whose bounds are not specified in the declaration of the array, but which *assumes* the same shape as the corresponding actual array argument. The array specification for such an array takes one of the forms:

  type, DIMENSION(*lower_bnd* :) :: *list of names*
  type, DIMENSION (:) :: *list of names*

  The second form is equivalent to the first with a lower bound equal to 1. In both cases the upper bound will only be established on entry to the procedure, and will be whatever value is necessary to ensure that the extent of the dummy array is the same of the actual array argument.
- Example:

```
   ...
   REAL, DIMENSION(10:30) :: a, b
   ...
   CALL array_example(a,b)
   ...

   SUBROUTINE array_example(dum_arr_1, dum_arr_2)
      IMPLICIT NONE
      REAL, DIMENSION(:) :: dum_arr_1, dum_arr_2
```

```
   ...
END SUBROUTINE array_example
```

- A call or reference to a procedure which has an assumed-shape dummy argument is one of the situations in which the calling program unit *must* have full details about the interface of the called procedure, in other words, the procedure must have an *explicit interface* available at the point of the call. This can be achieved by placing the procedure in a module.
- The intrinsic procedures

  SIZE(*array_name*)
  LBOUND(*array_name,1)*
  UBOUND(*array_name,1*)

  return the size, the suffix lower bound and the suffix upper bound of the array *array_name*.
- An **automatic array** is an explicit-shape array in a procedure, which is *not* a dummy argument array, which has non-constant bounds, and which obtains the information required to calculate its bounds from outside the procedure at the time of entering the procedure.
- There are only three situations in which an explicit-shape array may have non-constant bounds:
  - if the array is a dummy argument to a procedure;
  - if the array is an automatic array in a procedure;
  - if the array is the result of a function.
- An **array valued function** must have the bounds of the array-valued result variable declared in a type declaration statement within the body of the function subprogram.

  FUNCTION *name*(...)
  IMPLICIT NONE
  REAL, DIMENSION(*dim*) :: *name*

  For example:

  FUNCTION *name*(*arr*, ...)
  IMPLICIT NONE
  REAL, DIMENSION(:) :: *arr*
  REAL, DIMENSION(SIZE(*arr*)) :: *name*
- Derived type definitions may have arrays as components, provided that they are explicit-shape arrays having constant bounds.

Fortran 90 syntax introduced in Chapter 7

Array declaration	*type*, DIMENSION(*extent*) :: *list of names* *type*, DIMENSION(*lower_bnd* : *upper_bnd*) :: *list of names* *type*, DIMENSION(*lower_bnd* :) :: *list of names* *type*, DIMENSION(:) :: *list of names*
Array element	*array_name*(*integer_expression*)
Array constructor	(/ *list of values* /) (/ (*value_list, int_var = initial, final, inc*) /)
Array input/output	READ *, *array_element, array_name* PRINT *, (*array_element_list, int_var = initial, final, inc*) ⋮
Whole array operations	a=b*c etc., where a, b and c are conformable arrays

Example 7.1x

Problem (7.1x)

Write a program to calculate: (1) the dot product of two three-dimensional vectors; (2) the vector product of two three-dimensional vectors; (3) the scalar triple product of three three-dimensional vectors.

Analysis (7.1x)

Consider the two three-dimensional vectors

$$\boldsymbol{a} = (a_1, a_2, a_3), \qquad \boldsymbol{b} = (b_1, b_2, b_3).$$

The dot product of the two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is the scalar $\boldsymbol{a} \cdot \boldsymbol{b}$ defined by:

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1\, b_1 + a_2\, b_2 + a_3\, b_3$$

The vector product of the two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is the vector $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ defined by:

$$\boldsymbol{c} = (a_2\, b_3 - a_3\, b_2,\ a_3\, b_1 - a_1\, b_3,\ a_1\, b_2 - a_2\, b_1)$$

The scalar triple product of the three vectors $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$ is the scalar $[\boldsymbol{abc}]$ defined by:

$$[\boldsymbol{abc}] = \boldsymbol{a} \cdot (\boldsymbol{b} \times \boldsymbol{c})$$

Structure plan:

1	Read *selector* to choose the product required
2	Read two vectors in cases (1) and (2) and three vectors in case (3)
3	Calculate required product
4	Print result

Solution (7.1x)

```
MODULE products
   IMPLICIT NONE
   SAVE
   !
   CONTAINS
   !
   !
   FUNCTION dot_product(a, b)
      IMPLICIT NONE
      REAL, DIMENSION(3), INTENT(IN) ::  a, b
      REAL :: dot_product
      !
      dot_product = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
   END FUNCTION dot_product
   !
   FUNCTION vector_product(a, b)
      IMPLICIT NONE
      REAL, DIMENSION(3), INTENT(IN) :: a, b
      REAL, DIMENSION(3) :: vector_product
      !
      vector_product(1) = a(2)*b(3) - a(3)*b(2)
      vector_product(2) = a(3)*b(1) - a(1)*b(3)
      vector_product(3) = a(1)*b(2) - a(2)*b(1)
   END FUNCTION vector_product
   !
END MODULE products
```

```
PROGRAM test_products
   USE products
   IMPLICIT NONE
   REAL, DIMENSION(3) :: a, b, c, vp
   REAL :: dp, stp
   INTEGER :: selector
   !
   PRINT*, "selector = ? (1(dp), 2(vp), 3(stp))"
   READ*, selector
   !
   CASE SELECT (selector)
   CASE (1)
      PRINT*, "a = ?"
      READ*,   a(1), a(2), a(3)
      PRINT*, "b = ?"
      READ*,   b(1), b(2), b(3)
      dp = dot_product(a, b)
      PRINT*, "a . b = ", dp
   CASE (2)
      PRINT*, "a = ?"
      READ*,   a(1), a(2), a(3)
      PRINT*, "b = ?"
      READ*,   b(1), b(2), b(3)
      vp = vector_product(a, b)
      PRINT*, "a x b = ", vp
   CASE (3)
      PRINT*, "a = ?"
      READ*,   a(1), a(2), a(3)
      PRINT*, "b = ?"
      READ*,   b(1), b(2), b(3)
      PRINT*, "c = ?"
      READ*,   c(1), c(2), c(3)
      vp = vector_product(b, c)
      stp = dot_product(a, vp)
      PRINT*, "a . b x c = ", stp
   END SELECT
   !
END PROGRAM test_products
```

CHAPTER 8. MORE CONTROL OVER INPUT AND OUTPUT

OVERVIEW

The input and output facilities of any programming language are extremely important, because it is through these features of the language that communication between the user and the program is carried out. However, this frequently leads to a conflict between ease of use and complexity and Fortran 90, therefore, provides facilities for input and output at two quite different levels.

The list-directed input and output statements that we have been using up to now provide the capability for straightforward input from the keyboard and output to the printer. These statements, however, allow the user very little control over the source or the layout of the input data, or over the destination or layout of the printed results.

This chapter introduces the more general input/output features of Fortran 90, by means of which the programmer may specify exactly how the data will be presented and interpreted, from which of the available input units it is to be read, exactly how the results are to be displayed, and to which of the available output units the results are to be sent. Because of the interaction with the world outside the computer, input and output has the potential for more execution-time errors than most other parts of a program, and Fortran's approach to the detection of such errors is also briefly discussed.

SUMMARY

- The data for a list-directed READ statement is a sequence of alternating values and **value separators**, each of which may be a comma, a slash (/), a blank, or the end of the record (that is, of the line), preceded and/or followed by any number of consecutive blanks.
- If there is no value between two consecutive value separators then a **null value** is read, leaving the corresponding variable in the input list unchanged.
- Character data read by a list-directed READ statement must be delimited by matching **apostrophes** or **quotation marks** unless it is contained on a single line, does not contain any blanks, commas or slashes, does not begin with an apostrophe or quotation mark, and does not begin with a sequence of digits followed by an asterisk.
- For a great many purposes list-directed input is perfectly satisfactory.
- The programmer has virtually no control over the layout of the results printed by a list-directed PRINT statement.
- The READ and PRINT statements actually have three forms:

  READ *ch_var, input_lis*
  READ *label, input_lis*
  READ *, *input_lis*

  PRINT *ch_var, output_lis*
  PRINT *label, output_lis*
  PRINT *, *output_lis*

- The item following the keyword (READ or PRINT) is a **format specifier** which provides a link to the information necessary for the required editing to be carried out as part of the input or output process. This information is called a **format** and consists of a list of **edit descriptors** enclosed in parenthesis:

  (edit_descriptor_list)

- The first variation, in which the format specifier *ch_var* is a character expression (a character constant, a character variable, a character array, a character array element) is called an **embedded format** because the format itself appears as part of the READ or PRINT statement.

  READ '(*edit_descriptor_list*)', *input_list*

  or

  READ "(*edit_descriptor_list*)", *input_list*

- In the second variation the statement label is the label of a new type of statement called a FORMAT **statement** which contains the appropriate format.

- In the third variation, the asterisk indicates that the format to be used is a list-directed format which will be created by the processor to meet the perceived needs of the particular input or output list – hence its name.

- The edit descriptors that are used for input in conjunction with a READ statement are shown below. They fall into two categories: those concerned with the editing of actual data (the first six), and those concerned with altering the order in which the characters in the input record are edited (the last four).

Descriptor	*Meaning*
Iw	Read the next w characters as an integer
F$w.d$ E$w.d$	Read the next w characters as a real number with d digits after the decimal place if no decimal point is present
Aw	Read the next w characters as characters
A	Read sufficient characters to fill the input list item, stored as characters
Lw	Read the next w characters as the representation of a logical value
nX	Ignore the next n characters
Tc	Next character to be read is at position c
TLn TRn	Next character to be read is n charaters before (TL) or after (TR) the current position

- If the data is typed with a decimal point then the value of d is irrelevant (although it must be included in the format).

- As a character variable has a defined length any string which is to be stored in it must be made to have the same length. If we assume that length of the input list item is *len* then the following rules apply:

  - If w is less than *len* then extra blank characters will be added at the end so as to extend the length of the input character string to *len*. This is similar to the situation with assignement.

  - If w is greater than *len*, however, the rightmost *len* characters of the input character string will be stored in the input list item. This is *opposite* of what happens with assignement!

- An A edit descriptor without any field width w is treated as though the field width was identical to the length of the corresponding input list item.
- The edit descriptor Lw processes the next w characters to derive either a **true** value, a **false** value, or an error. There are exactly two ways of representing **true** and **false** in the data, namely as a string of characters in one of the following forms, optionally preceded by one or more spaces:

  T*cc...c* or .T*cc...c*
  F*cc...c* or .F*cc...c*

  where c represents any character. The lower case letters t and f are treated as being equivalent to the upper case letters T and F in a logical input field.

- Example: List of input data: 123456789

```
READ '(I9)', n
                                          ! n=123456789
READ '(I3,I3,I3)', n1, n2, n3
                                          ! n1=123; n2=456; n3=789
READ '(4X,I5)', num
                                          ! num=56789
READ '(I2,3X,I3)', i, j
                                          ! i=12; j=678
READ (T4,I2,T8,I2,T2,I4)', x, y, z
                                          ! x=45; y=89; z=2345
READ '(F3.1,F2.2,F3.0,TL6,F4.2)', r1, r2, r3, r4
                        ! r1=12.3; r2=0.45; r3=678.0; r4=34.56
```

- A FORMAT statement is a special, labelled, non-executable statement which takes the form

  label FORMAT (*edit_descriptor_list*)

  It may appear anywhere in the program unit after the initial statement and any USE statements and before the END statement; it must also come before any internal procedures.

- The FORMAT statements in a program unit should be kept together for ease of reference, either before or after the executable statements in the program unit.
- The main edit descriptors that are available for output are shown below:

Descriptor	*Meaning*
Iw	Output an integer in the next w character positions
F$w.d$	Output a real number in the next w character positions with d decimal places
E$w.d$	Output a real number in the next w character positions using an exponential format with d decimal places in the mantissa and four characters for the exponent
Aw	Output a character string in the next w character positions
A	Output a character string, starting at the next character position, with no leading or trailing blanks
Lw	Output w - 1 blanks, followed by T or F to represent a logical value
nX	Ignore the next n character positions
Tc	Output the next item starting at character position c
TLn	Output the next item starting n character positions
TRn	before (TL) or after (TR) the current position
"$c_1c_2\ldots c_n$"	Output the string of characters $c_1c_2\ldots c_n$
'$c_1c_2\ldots c_n$'	starting at the next character position

- A real constant may be written followed by an exponent and a similar format is allowed for numbers being input by a READ statement. In this case the exponent may take one of three forms:
  - ◇ a signed integer constant;
  - ◇ E followed by an optionally signed constant;
  - ◇ D followed by an optionally signed constant.

  In the latter two cases the letter (D or E) may be followed by one or more spaces.
- If the exponent is greater than 99, or less than -99, then the exponent will be output as a plus or minus sign, followed by a three digit exponent. Some Fortran 90 processors may choose to use this form of representation for all values of the exponent.
- For *all* numeric edit descriptors, if the number does not require the full field width w it will be preceded by one or mores spaces.
- Example: Consider the real number x = 0.0000361764

```
PRINT '(F10.4)', x     ! will print   ####0.0000
PRINT '(F12.6)', x     !              ####0.000036
PRINT '(F14.8)', x     !              ####0.00003618
PRINT '(E10.4)', x     !              0.3618E-04
PRINT '(E12.6)', x     !              0.361764E-04
PRINT '(E14.8)', x     !              0.36176400E-04
```

- Considering the Aw edit descriptor, we need to establish exactly what happens if the length of the output list item is not exactly w. The rules that apply here are similar to those that we had for input, where *len* is the length of the character variable or constant being output:

  - If w is greater than *len* then the character string will be right-justified within the output field, and will be preceded by one or more blanks. This is similar to what happens with the I, F and E edit descriptors.
  - If w is less than *len* then the leftmost w characters will be output.

- A number, called a **repeat count**, may be placed *before* the I, F, E, A or L edit descriptors to indicate how many times they are to be repeated.
- Example:

```
(I5,I5,I5,F6.2,F6.2,F6.2,F6.2)
                                        ! is the same as
(3I5, 4F6.2)
```

- A format may also include a **character constant edit descriptor**, which takes the form of a character constant, and is output at the next character position.
- Example:

```
PRINT '("The result is ",I5)', result
                                              ! is the same as
PRINT "('The result is ',I5)", result
```

- READ and WRITE statements with control information lists are used to provide greater flexibility than is possible with the simple READ and PRINT statements which always use the default input and output units:

  READ (*cilist*) *input_list*
  WRITE (*cilist*) *output_list*

- The **control information list** *cilist* consists of one or more items, known as **specifiers**, separated by commas. They all take the same basic form:

  keyword = *value*

- There must always be a **unit specifier** in the control information list which is used to specify the input or output unit to be used for a READ or WRITE statement. It takes the form

  UNIT = *unit*

  where *unit* is the input device from which input is to be taken. It either takes the form of an integer expression whose value is zero or positive, or it may be an asterisk to indicate that the default input or output unit is to be used.

- The choice of numbers for the default input and output units is dependent upon the particular implementation. In this book *we shall assume that the default input unit is* 5 *and that the default output unit is* 6, and therefore that:

```
READ (UNIT = 5) a       ! is equivalent to
READ (UNIT = *) a
                        ! and
WRITE (UNIT = 6) a      ! is equivalent to
WRITE (UNIT = *) a
```

- A **format specifier** is used to specify the format to be used with a READ or WRITE statement. It takes one of the forms

  FMT = *ch_var*
  FMT = *label*
  FMT = *

- The IOSTAT specifier is concerned with monitoring the outcome of the reading process, and takes the form

  IOSTAT = *io_status*

  where *io_status* is an integer variable. At the conclusion of the execution of the READ statement *io_status* will be set to a value which the program can use to determine whether any errors occurred during the input process. There are four possibilities:

  - the variable is set to zero to indicate that no errors occurred;
  - the variable is set to a processor-dependent positive value to indicate that an error has occurred;
  - the variable is set to a processor-dependent negative value to indicate that a condition known as end-of-file condition has occurred;
  - the variable is set to a processor-dependent negative value to indicate that a condition known as end-of-record condition has occurred.

- The first character of each output record being sent to the unit designated by the processor as a printer is removed before printing takes place and used to control vertical printer movements; it is called the **printer control character**.

Character	*Vertical spacing before printing*
# (space)	one line
0 (zero)	two lines
1 (one)	first line of next page
+ (plus)	no paper advance (overprint)

  Because the first character is removed and not printed it is important that we insert an extra (control) character at the start of each record that is to be output to the printer.

- Formats, or parts of formats, are repeated as many times as required until the input or output list has been exhausted. (. . .)

- The slash (/) edit descriptor indicates the end of the current record. It can be used to define a format which processes two or more separate lines, or (more accurately) **records**. On input, a / causes the rest of the current record to be ignored and the next input item to be the first item of the *next* record. On output, a / terminates the current record and starts a new one. The / edit descriptor can, but need not be, separated from any preceding or succeeding descriptor by a comma. Multiple consecutive / descriptors cause input records to be skipped or null (blank) records to be output.

- Example:

```
READ '(3F8.2/3I6)', a, b, c, p, q, r
```

  will read three real numbers from the first record and three integers from the second.

- Example:

```
WRITE (UNIT=6, FMT=201) a, b, a+b, a*b
201 FORMAT("1", T10, "Multi-record example"/                    &
           "0", "The sum of", F6.2, " and", F6.2, " is", F7.2/ &
           1X, "Their product is", F10.3)
```

  will cause the following output:

```
-------------------------------------------------- (new page}
         Multi-record example

The sum of 12.25 and 23.50 is  35.75
Their product is   287.875
```

- Example:

```
READ '(3F8.2//3I6)', a, b, c, p, q, r
```

  will read three real numbers from the first record and three integers from the third.

- Example:

```
WRITE (UNIT=6, FMT=202) a, b, a+b, a*b
202 FORMAT("1" / T10, "Multi-record example"//                &
        "0", "The sum of", F6.2, " and", F6.2, " is", F7.2// &
        1X, "Their product is", F10.3)
```

  will cause the following output:

```
-------------------------------------------------- (new page}

         Multi-record example


The sum of 12.25 and 23.50 is  35.75

Their product is   287.875
```

Fortran 90 syntax introduced in Chapter 8	
Input/output statements	READ (*cilist*) *input_list* WRITE (*cilist*) *output_list*
Format specifier	(*list of edit descriptors*)
FORMAT statement	*label* FORMAT(*list of edit descriptors*)
Edit descriptors	Iw, F$w.d$, E$w.d$, Aw, A, Lw nX, Tc, TLn, TRn, /
Control information list specifiers	UNIT = *unit* UNIT = * FMT = *label* FMT = '*format_specifier*' FMT = * IOSTAT = *int_var*

CHAPTER 9. USING FILES TO PRESERVE DATA

OVERVIEW

One of the most important aspects of computing is the ability for a program to save the data that it has been using for subsequent use either by itself or by another program. This involves the output of the data to a file, usually on some form of magnetic or optical medium, for input at some later time. Files may be written and read sequentially or, on some types of media, the information in a file may be written and read in a random order. In either case the file may be stored permanently within the computer system, for example on a magnetic or magneto-optical disk which is an integral part of the computer, or it may be stored on some medium, such as disk or tape, which may be removed from the computer either for safe-keeping of for physical transport to another computer.

This chapter shows how the READ and WRITE statements discussed in Chapter 8 can be used to read data from a file and write data to a file, in a sequential manner, and introduces several additional statements which are required when dealing with files. More sophisticated uses of files, including random access to information stored in a file, is discussed later, in Chapter 15.

SUMMARY

- Information that is to be preserved after the execution of a program is ended is stored in a **file**.
- A file consists of a sequence of **records**.
- The records in a file may be accessed in a **sequential** manner, or in a **random access** manner.
- A file may consist of **formatted records** and, optionally, **one endfile record**, or it may consist of **unformatted records**, and optionally, **one endfile record**.
- A **formatted record** consists of a sequence of characters selected from those which can be represented by the processor being used.
- A formatted record is written by a formatted WRITE statement, or by some means external to Fortran; it is read by a formatted READ statement.
- Example:

```
WRITE (UNIT = 7, FMT = 200) var_1, var_2, var_3
WRITE (UNIT = *) var_1, var_2, var_3
```

- An **unformatted record** consists of a sequence of values (in a processor-dependent form) and is, essentially, a copy of some part, or parts, of the memory of the computer.
- An unformatted record is written by an unformatted WRITE statement, and it is read by an unformatted READ statement. These are the same as the corresponding formatted statements but without any format specifier.
- Example:

```
WRITE (UNIT=9) var_1, var_2, var_3
WRITE (UNIT=3, IOSTAT=ios) x, y, x
READ (UNIT=9) var_4, var_5, var_6
READ (UNIT=3, IOSTAT=io_status) a, b, c
```

- One important difference between the input/output of formatted and unformatted records is that whereas a formatted input or output statement may read or write more than one record by use of a suitable format, for example

  ```
  WRITE (UNIT=3, FMT='(2I8/(4F12.4))') int_1, int_2, arr
  ```

  an unformatted input or output statement will always read or write exactly *one* record. The number of items in the input list of an unformatted READ statement must therefore be the same as the number in the output list of the unformatted WRITE statement that wrote it, or fewer (in which case the last few items in the record are ignored).

- An **endfile record** is a special type of record which can only occur as the last record of a file and as is written by a special statement of one of the forms

  ENDFILE *unit*

  or

  ENDFILE (*auxlist*)

  In the first case *unit* is the output unit to which an endfile record is to be written, while in the second case *auxlist* consists of a UNIT specifier and, optionally, an IOSTAT specifier, where these specifiers are the same as those already introduced for use with a WRITE statement.

- An ENDFILE statement writes a special endfile record to the specified file and leaves the file positioned after that record. Any information which physically exists after an endfile record becomes inaccessible thereafter, and may be considered to have been deleted. It is not subsequently possible to write to, or read from, that file without first repositioning it by using either a REWIND or a BACKSPACE statement.

- If an endfile record is read by an input statement it will cause an **end-of-file condition** which can be detected by an IOSTAT specifier in a READ statement. *If it is not specifically detected in this way an error will occur and the program will fail.*

- It is a good practice to place an endfile record at the end of all sequential files in order that a program which subsequently reads the file can easily detect the end of the information in the file without the need for any other special records or counts. It also acts as a safeguard against an error which might cause the program not to detect the end of the information in the file.

- For any information to be transferred between a file and a program the file must be **connected** to a unit; in other words, a logical connection, or relationship, must be established between the file and the unit number that will be used in any READ or WRITE statements which are to use that file. This connection is initiated by means of an OPEN statement, which takes the form

  OPEN (*open_specifier_list*)

  where *open_specifier_list* is a list of specifiers, some of which are shown below:

UNIT = *unit_number*
FILE = *file_name*
STATUS = *file_status*
FORM = *format_mode*
ACTION = *allowed_actions*
POSITION = *file_position*
IOSTAT = *ios*

- The UNIT specifier must be present, and takes the same form as in the READ, WRITE and ENDFILE statements. All the remaining specifiers are optional and enable use to specify various requirements regarding the file that is to opened and to monitor the opening process itself.
- The FILE specifier is used to specify the name by which a file is known to the computer system. The character expression *file_name* takes the form of a filename for the particular computer system, after the removal of any trailing blanks.
- The STATUS specifier is used to define certain restrictions on our use of the file. The character expression *file_status* must take one of the values OLD, NEW, REPLACE, SCRATCH or UNKNOWN, after the removal of any trailing blanks.
- The FORM specifier is used to specify if the records in the file must either all be formatted or all be unformatted. The character expression *format_mode* must take one of the values FORMATTED or UNFORMATTED, after the removal of any trailing blanks.
- Example:

```
OPEN (UNIT=7, FILE="datafile", STATUS="OLD", FORM="FORMATTED")
```

- The ACTION specifier is used to specify what type of input/output actions are allowed with the file. The character expression *allowed_actions* must take one of the three values READ, WRITE or READWRITE, after the removal of any trailing blanks.
- The POSITION specifier allows the programmer to instruct the OPEN statement to cause the file to be positioned at some point other than the beginning. The character expression *file_position* must take one of the three values REWIND, APPEND or ASSIS, after the removal of any trailing blanks.
- The specifier IOSTAT is concerned with recognizing when an error occurs during the connection process and operates in the same way as has already benn discussed in connection with the READ, WRITE and ENDFILE statements. In the event of an error during the opening of a file the execution of the program will be terminated unless it is detected by the program.
- Fortran provides two additional **file-positioning statements** to alter the position in a file without reading or writing any records.
- The first of these file-positioning statements

  BACKSPACE *unit_number*

  or

  BACKSPACE (*auxlist*)

causes the file to be positioned just before the *preceding* record (that is, it enables the program to read the immediately previously read record again).

- The other file-positioning statement

  REWIND *unit_number*

  or

  REWIND (*auxlist*)

  causes the file to be positioned just before the *first* record so that a subsequent input statement will start reading or writing from the beginning.

- Writing a record to a sequential file destroys all information in the file after that record.

Fortran 90 syntax introduced in Chapter 9

File connection statement	OPEN (*open_specifier_list*)
Unformatted input/ output statements	READ (*control_information_list*) *input_list* WRITE (*control_information_list*) *input_list* where the *control_information_list* does not include a format specifier
ENDFILE statement	ENDFILE *unit* ENDFILE (*auxlist*)
File positioning statements	BACKSPACE *unit* BACKSPACE (*auxlist*) REWIND *unit* REWIND (*auxlist*)
Control information list specifiers	FILE = *file_name* STATUS = *file_status* where *file_status* is one of "OLD", "NEW", "REPLACE", "SCRATCH" or "UNKNOWN" FORM = *format_mode* where *format_mode* is either "FORMATTED" or "UNFORMATTED" ACTION = *allowed__actions* where *allowed_actions* is one of "READ", "WRITE" or "READWRITE" POSITION = *file_position* where *file_position* is one of "REWIND", "APPEND" or "ASIS"

Example 9.1x

Problem (9.1x)

Write a program which will read a file containing book references, will select those that contain a character string specified by the user and will print the selected references to another file.

Analysis (9.1x)

Structure plan:

1	Read unit numbers *in_unit* and *out_unit*
2	Read file names *in_file* and *out_file*
3	Read *string*
4	Use intrinsic function LEN_TRIM to obtain the length of *string*, without counting any trailing blank characters.
5	Call subroutine *read_write* to perform the required selection.

The subroutine *read_write* will depend on the form of the file *in_file* that contains the book references. In the example shown it is supposed that each book reference is composed by a variable number of 80 character lines, the last line terminated by the character string "//". Book references are separated by a blank line. Only the first line of each book reference is searched for the string *string*.

Solution (9.1x)

```
      PROGRAM books
!--------------------
!        FJR / 2.NOV.1994 / 22.OUT.2003
!--------------------
      IMPLICIT NONE
      CHARACTER (LEN=15) :: in_file, out_file
      CHARACTER (LEN=10) ::  string
      INTEGER :: in_unit, out_unit, len_string
!--------------------
      PRINT *, "input unit, output unit = ?"
      READ *, in_unit, out_unit
!--------------------
      PRINT *, "input file, output file = ?"
      READ *, in_file, out_file
!--------------------
      OPEN (UNIT=in_unit, FILE=in_file , STATUS="OLD")
      OPEN (UNIT=out_unit, FILE=out_file, STATUS="NEW")
!--------------------
      PRINT *, "string = ? (1 <= LEN(string) <= 10"
      READ *, string
!--------------------
      len_string=LEN_TRIM(string)
      CALL read_write(string, len_string, in_unit, out_unit)
!--------------------
      END PROGRAM books
```

```
      SUBROUTINE read_write(string, len, in, out)
      IMPLICIT NONE
      CHARACTER (LEN=80) :: c1, c2
      CHARACTER (LEN=len) ::  string
      CHARACTER (LEN=2), PARAMETER :: cs="//"
      INTEGER :: in, out, len, ix, iy
!-------------------------
      DO
         READ (UNIT=in, FMT=100) c1
         IF (c1(1:7) == "ENDFILE") RETURN
         ix=INDEX(c1,string)
         IF (ix == 0) CYCLE
         WRITE (UNIT=out, FMT=100) TRIM(c1)
!--------------------
         DO
            READ (UNIT=in, FMT=100) c2
            WRITE (UNIT=out, FMT=100) TRIM(c2)
            iy = INDEX(c2,cs)
!---------------
            IF (iy > 0) THEN
               READ (UNIT=in, FMT=100) c2
               WRITE (UNIT=out, FMT=100) TRIM(c2)
               EXIT
            END IF
!---------------
         END DO
!--------------------
      END DO
!-------------------------
  100 FORMAT(A)
      END SUBROUTINE read_write
```

CHAPTER 10.
AN INTRODUCTION TO NUMERICAL METHODS IN FORTRAN 90 PROGRAMS

OVERVIEW

The main area of application for Fortran programs is, and always has been, the solution of scientific and technological problems – a process which usually involves the solution of mathematical problems by numerical, as opposed to analytical, means.

This chapter introduces some of the major limitations that are imposed on numerical problem solving by the physical characteristics of computers, as well as by the nature of the problems being solved, and the means that are provided in Fortran 90 to ensure that the effects of these constraints are both predictable and controllable. Two of the most common numerical problems, the fitting of a straight line through a set of experimental or empirical data and the solution of non-linear equations, are then discussed, and examples given of how these problems may be solved in Fortran.

For those particularly interested in this aspect of programming, Chapter 18 will return to the subject in rather more detail, with examples of several other commonly required numerical methods.

SUMMARY

- REAL numbers are stored in a computer as **floating-point** approximations to their true mathematical values. The accuracy of this approximation is determined by the form of the floating point number which is allocated a fixed number of **bits** for the **mantissa** (thus defining the *precision*) and a fixed number for the **exponent** (thus defining the *range* of the numbers).
- All REAL calculations are subject to **round-off errors**, and the programmer must take care to perform complicated calculations in such a way as to minimize these effects.
- **Overflow** will occur if a calculation would result in an exponent for a real number being larger than the maximum possible exponent allowed. Overflow results in an error condition.
- **Underflow** will occur if a calculation would result in an exponent for a real number being smaller than the minimum possible exponent allowed. The result of underflow is that the result of the calculation is treated as zero; it is not treated as an error by many processors.
- To permit more precise control over the precision and exponent range of floating point numbers REAL variables are **parameterized**. That is, they have a parameter associated with them that specifies *minimum precision and exponent range* requirements. This is called the **kind type parameter**. When this parameter is not specified explicitly, the type of the floating-point number is said to be **default-real**. The kind type parameter value assigned to a default real is processor-dependent.
- Example:

```
REAL :: a, b, c, d                ! default-real
```

```
REAL, DIMENSION(10) :: x, y     ! default-real
REAL :: p(20), q(40), r(60)     ! default-real

REAL(KIND=4) :: e, f
REAL(KIND=1) :: g, h
REAL(KIND=4), DIMENSION(10) :: u, v
REAL(KIND=2) :: s(8), t(5)
```

- For any variable or constant that is an intrinsic type, the value of its kind type can be found by using the intrinsic function KIND.
- Example:

```
REAL(KIND=3) :: x
REAL :: y
INTEGER :: i, j

i = KIND(x)     ! i = 3
j = KIND(y)     ! j = value for the kind type of
                !     a default real number
```

- The SELECTED_REAL_KIND intrinsic function may be used to determine the kind type parameter of the real number representation on the current processor which meets, at least, a specified degree of precision and exponent range.
- This intrinsic function has two optional arguments P and R: P is a scalar integer argument specifying the minimal number of decimal digits required; R is a scalar integer argument specifying the minimal decimal exponent range required. The result of the function is the kind type that meets, or minimally exceeds, the requirements specified by P and R. If more than one kind type parameter meets the requirements, the value returned is the one with the smallest decimal precision. If the precision is not available the result is -1, if the range is not available is -2, if neither is available is -3.
- Example:

```
REAL(KIND=SELECTED_REAL_KIND(P=8, R=30)) :: m
REAL(KIND=SELECTED_REAL_KIND(P=6, R=30)) :: n
```

- The important point to notice here is that, regardless of the computer on which the above code is compiled and executed, it will not have to be changed in any way to meet the specified precision and range requirements. The values returned by the SELECTED_REAL_KIND function may change, but that is of no consequence to the program as far as portability is concerned. In fact, because of the lack of portability of the kind type parameter values, we recommend that they *only* be used via the SELECTED_REAL_KIND function. The easiest way to do this is to define a constant for use in subsequent variable declarations.
- Example:

```
INTEGER, PARAMETER :: real_8_30 = SELECTED_REAL_KIND(P=8, R=30)
...
REAL (KIND=real_8_30) :: x, y, z
```

- It is permissible to not specify a value for R in a reference to SELECTED_REAL_KIND, in which case the range provided will be the default range for the precision specified.
- Example:

```
INTEGER, PARAMETER :: real_8 = SELECTED_REAL_KIND(P=8)
```

- Real constants also have a kind type parameter and, as with variables, if none is specified then the constant is of type default real. The kind type parameter is explicitly specified by following the constant's value by an underscore and the kind parameter.
- Example:

```
-103.4_7        ! Real of kind type 7
3.14_high       ! Real of kind type high
4.0E7_2         ! Real of kind type 2
2.7             ! Default real (processor-dependent kind type)
```

- The use of parameterized REAL variables and constants, in conjunction with the SELECTED_REAL_KIND intrinsic function, provides a portable means of specifying the precision and exponent range for numerical algorithms.
- A **well-conditioned** problem is one which is relatively insensitive to changes in the values of its parameters, so that small changes in these parameters only produce small changes in the output. An **ill-conditioned** problem is one which is highly sensitive to changes in its parameters, and where small changes in these parameters produce large changes in the output.
- A numerical process (algorithm) is said to be **stable** if the answer it gives is the mathematically exact answer to a problem that is only slightly different from the problem given. It is said to be **unstable** if the answer it provides is to a problem substantially different from the one given.
- The two principal causes of unstable algorithms are **round-off error** and **truncation error**.

Fortran 90 syntax introduced in Chapter 10

Variable declarations	REAL(KIND=*kind_type*) ... :: *list of variable names*
Literal constant definition	*numerical_literal_kind_type*

Example 10.2

Problem (10.2)

Write a program to find the root of the equation $f(x) = 0$ which lies in a specified interval. The program should use an external function to define the equation, and the user should input the details of the interval in which the root lies and the accuracy required.

Analysis (10.2)

We have already discussed the mathematics underlying the method, and so can proceed directly to the design of our program.

Structure plan:

1 Read range (*left* and *right*), *tolerance* and *maximum_iterations*
2 Call subroutine *bisect* to find a root in the interval (*left, right*)
3 If root found then
  3.1 Print root
  otherwise
  3.2 Print error message

Subroutine *bisect*
Real dummy arguments: *xl_start, xr_start, tolerance, zero, delta*
Integer dummy arguments: *max_iterations, num_bisecs, error*

[Note that *zero* is the root, *delta* is the uncertainty in the root (it will not exceed *tolerance*), *num_bisecs* is the number of interval bisections taken and *error* is a status indicator]

1 If *xl_start* and *xr_start* do not bracket a root then
  1.1 Set *error* = - 1 and return
2 Set *x_left=xl_start, x_right=xr_start*
3 Repeat *max_iterations* times
  3.1 Calculate mid-point (*x_mid*) of interval
  3.2 If (*x_mid* - *x_left*) $\leq$ *tolerance* then exit with
    zero = *x_mid*, *delta* = *x_mid* - *x_left*,
    and *error* = *0* to indicate success
  3.3 Otherwise, determine which half interval the root lies in
    and set *x_left* and *x_right* appropriately
4 No root found so set *error* = -2 to indicate failure to converge quickly enough

3.3.1 If *f(x_left)* $\times$ *f(x_mid)* < 0 then
  3.3.1.1 set *x_right* to *x_mid*
  otherwise
  3.3.1.1 set *x_left* to *x_mid*

Solution (10.2)

```
MODULE constants
   IMPLICIT NONE
   !
   ! Define a kind type q to have at least 6 decimal digits
   ! and an exponent range form 10**30 to 10**(-30)
   INTEGER, PARAMETER :: q = SELECTED_REAL_KIND(P=6, R=30)
END MODULE constants

PROGRAM zero_find
   USE constants
   IMPLICIT NONE
   !
   ! This program finds a root of the equation f(x)=0 in a
   ! specified interval to within a specified tolerance of
   ! the true root, by using the bisection method
   !
   ! Input variables
   REAL(KIND=q), EXTERNAL :: f
   REAL(KIND=q) :: left, right, tolerance
   INTEGER :: maximum_iterations
   !
   ! Other variables
   REAL(KIND=q) :: zero, delta
   INTEGER :: number_of_bisections, err
   !
   ! Get range and tolerance information
   PRINT *, "Give the bounding interval (two values)"
   READ *, left, right
   !
   PRINT *, "Give the tolerance"
   READ *, tolerance
   !
   PRINT *, "Give the maximum number of iterations allowed"
   READ *, maximum_iterations
   !
   ! Calculate root by the bisection method
   CALL bisect(f, left, right, tolerance, maximum_iterations, &
               zero, delta, number_of_bisections, err)
   !
   ! Determine type of result
   SELECT CASE (err)
      CASE (0)
         PRINT *, "The zero is ", zero, "+- ", delta
         PRINT *, "obtained after ", number_of_bisections, &
                  " bisections"
```

```
      CASE (-1)
         PRINT *, "The input is bad"
      CASE (-2)
         PRINT *, "The maximum number of iterations has been exceeded"
         PRINT *, "The x value being considered was ", zero
   END SELECT
END PROGRAM zero_find

SUBROUTINE bisect(f, xl_start, xr_start, tolerance, max_iterations,&
                  zero, delta, num_bisecs, error)
   USE constants
   IMPLICIT NONE
   !
   ! This subroutine attempts to find a root in the interval
   ! xl_start to xr_start using the bisection method
   !
   ! Dummy arguments
   REAL(KIND=q), INTENT(IN) :: xl_start, xr_start, tolerance
   INTEGER, INTENT(IN) :: max_iterations
   REAL(KIND=q), INTENT(OUT) :: zero, delta
   INTEGER, INTENT(OUT) :: num_bisecs, error
   !
   ! Function used to define equation whose roots are required
   REAL(KIND=q), EXTERNAL :: f
   !
   ! Local variables
   REAL(KIND=q) :: x_left, x_mid, x_right, v_left, v_mid, v_right
   !
   ! Initialize the zero-bounding interval and the function
   ! values at the end points
   IF (xl_start < xr_start) THEN
      x_left = xl_start
      x_right = xr_start
   ELSE
      x_left = xr_start
      x_right = xl_start
   END IF
   !
   v_left = f(x_left)
   v_right = f(x_right)
   !
   ! Validity check
   IF (v_left * v_right >= 0.0  .OR.  tolerance <= 0.0  .OR.  &
      max_iterations < 1) THEN
      error = -1
      RETURN
   END IF
```

```
   !
   DO num_bisecs = 0, max_iterations
      delta = 0.5 * (x_right-x_left)
      x_mid = x_left + delta
      IF (delta < tolerance) THEN
         ! Convergence criteria satisfied
         error = 0
         zero = x_mid
         RETURN
      END IF
      !
      v_mid = f(x_mid)
      ! ************************************************************
      ! Remove the following print statement when the program
      ! has been thoroughly tested
      PRINT '("Iteration", I3, 4X, 3F12.6, "   (", F10.6, ")")', &
         num_bisecs, x_left, x_mid, x_right, v_mid
      ! ************************************************************
      IF (v_left * v_mid < 0.0) THEN
         ! A root lies in the left half of the interval
         ! Contract the bounding interval to the left half
         x_right = x_mid
         v_right = v_mid
      ELSE
         ! A root lies in the right half of the interval
         ! Contract the bounding interval to the right half
         x_left = x_mid
         v_left = v_mid
      END IF
   END DO
   !
   ! The maximum number of iterations has been exceeded
   error = -2
   zero = x_mid
END SUBROUTINE bisect

FUNCTION f(x)
   USE constants
   IMPLICIT NONE
   ! Function type
   REAL(KIND=q) :: f
   ! Dummy argument
   REAL(KIND=q), INTENT(IN) :: x
   f = x + EXP(x)
END FUNCTION f
```

CHAPTER 13. ARRAY PROCESSING AND MATRIX MANIPULATION

OVERVIEW

In Chapter 7 we discussed the basic principles of Fortran's array facilities in the context of rank-one arrays. In mathematical terms such arrays are suitable for representing vectors, but in order to represent matrices, or more complex rectangular structures, more than one subscript is required. The same general principles apply to rank-n arrays as were described earlier in the context of rank-one arrays, although the order of the array elements is occasionally important.

As well as extending the basic array concepts to rank-n arrays, however, Fortran contains several other powerful array features which are the subject of the major part of this chapter. These include dynamic arrays, whose shape is not determined until execution time, and sub-arrays, which are created from either a regular or an irregular set of elements of another array. Finally, the facilities for whole array processing are re-examined in the light of these new, more flexible, types of arrays, and additional concepts are introduced to add still further to the power of Fortran's array processing capability.

SUMMARY

- In Fortran, an array is formally defined as a compound entity that contains an ordered set of scalar entities, each one of the same type, arranged in a rectangular pattern.
- An array may have from one to seven dimensions. The **rank** of an array is defined as the number of its dimensions. A vector is a rank-one array; a matrix is a rank-two array.
- For each dimension there are two bounds which define the range of index values that are permitted for that dimension, the **lower** and the **upper index bounds.**
- The **extent** of a dimension is the number of permissible index values for that dimension. It is given by

  MAX(*upper_index_bound* - *lower_index_bound* +1, 0)
- An array may have any non-negative extent, *including zero*, for any of its dimensions.
- The **size** of an array is the total number of elements it contains and is equal to the product of its extents.
- The **shape** of an array is determined by the number of its dimensions and the extent along each dimension; it is representable as a rank-one array whose elements are the extents.
- There are four different classes of arrays: explicit-shape arrays, assumed-shape arrays, automatic arrays and deferred-shape arrays.
- **Explicit-shape arrays** are arrays whose index bounds for each dimension are specified when the array is declared in a type-declaration statement. In this context, specified does not necessarily mean fixed. It means that the index bounds can be calculated from information available when the arrays are declared.

- The rank and extent of each dimension of an **explicit-shape array** are specified by using the dimension attribute in a type declaration statement. Instead of the extent of one or more dimensions one may specify the lower and upper index bounds for the dimension.

  type, DIMENSION(*list of explicit-shape specifiers*) :: *list of names*

  The rank of the array is the number of *explicit-shape specifiers* given. Each of these specifies the lower and upper index bounds for one dimension and takes the form

  lower_bound : *upper_bound*

  or

  upper_bound

  where *lower_bound* and *upper_bound* are **specification expressions**. If the *lower-bound* is omitted it is taken to be 1.

- In the case of a rank-two array, that is, a matrix, the number of rows is specified first and the number of columns second.

- Example:

```
REAL, DIMENSION(8) :: a
INTEGER, DIMENSION(3, 10, 2) :: b
TYPE(point), DIMENSION(4, 2, 100, 8) :: c

REAL, DIMENSION(11:18) :: a
INTEGER, DIMENSION(5:7, -10:-1, 2) :: b
TYPE(point), DIMENSION(5:8, 0:1, 100, -3:4) :: c
```

- Explicit-shape arrays with constant index bounds can be specified in type declaration statements in either main programs or procedures. In a procedure, a dummy argument may be an explicit-shape array whose bounds are integer expressions, the values of which can be determined at the time of entry to the procedure.

- Example:

```
REAL, DIMENSION(15, 50) :: p
REAL, DIMENSION(15, 15, 2) :: q
...
CALL explicit(p, q, 15, 7)
...

SUBROUTINE explicit(a, b, m, n)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: m, n
   REAL, DIMENSION(m, n*n+1), INTENT(INOUT) :: a
   REAL, DIMENSION(-n:n, m, INT(m/n)), INTENT(OUT) :: b
   ...
END SUBROUTINE explicit
```

- The elements of an array form a sequence known as the **array element order**. It can ve visualized as all the elements of an array, of whatever rank, being arranged in a sequence in such a way that the first index of the element specification is varying most rapidly, the next index of the element specification is varying the second most rapidly, and continuing in this manner until the last index of the element specification is varying the least rapidly.
- An **array constructor** is a means of specifying a literal rank-one array-valued constant. It takes the form

  (/ *value_list* /)

  where each item in *value_list* is either a single value or a list in parentheses controlled by an implied DO.
- The RESHAPE intrinsic function constructs an array of rank greater than one and of a specified shape from the elements of a given rank-one array. This function has two arguments, both of which are rank-one arrays: the first is the source array and the second specifies the required shape. The elements of the source array are used *in array element order*.
- Example: The statement

  ```
  RESHAPE ( (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 /), (/ 2, 3 /) )
  ```

  produces the matrix

  $$\begin{bmatrix} 1.0 & 3.0 & 5.0 \\ 2.0 & 4.0 & 6.0 \end{bmatrix}$$
- Input and output of arrays, or parts of arrays, can be handled in three ways:
  - as a list of individual array elements;
  - as a list of array elements under the control of an implied DO;
  - as the complete array, by including the unsubscripted array name in the input or output list; in this case the array elements will be transferred *in array element order*.
- In general it is advisable to use an implied DO when reading or writing whole arrays as it makes abolutely clear in which order the data is to be presented or the results are to be printed.
- Example:

  ```
  REAL, DIMENSION(50,6) :: x

  PRINT '(6F8.2)', x    ! will print out the data as

  x(1,1)  x(2,1)  x(3,1)  x(4,1)  x(5,1)  x(6,1)
  x(7,1)  x(8,1)  x(9,1)  x(10,1) x(11,1) x(12,1)
  ...     ...     ...     ...     ...     ...
  x(49,1) x(50,1) x(1,2)  x(2,2)  x(3,2)  x(4,2)
  x(5,2)  x(6,2)  x(7,2)  x(8,2)  x(9,2)  x(10,2)
  ...     ...     ...     ...     ...     ...
  ```

```
PRINT '(6F8.2)', ((x(i,j), j=1,6), i=1,50)

                        ! will print out the data as

x(1,1)  x(1,2)  x(1,3)  x(1,4)  x(1,5)  x(1,6)
x(2,1)  x(2,2)  x(2,3)  x(2,4)  x(2,5)  x(2,6)
...     ...     ...     ...     ...     ...
```

- **Assumed-shape arrays** may only be dummy arguments of a procedure that has an explicit interface; they cannot occur in a main program. They take their shape from association with actual arguments when a procedure is referenced, hence the name assumed-shape. The actual argument must be of the same type and have the same rank as the dummy argument.
- The dimension attribute for an assumed-shape array takes the form

  DIMENSION (*list of assumed-shape specifiers*)

  The rank of the array is the number of *assumed-shape specifiers* given. Each *assumed-shape specifiers* specifies the lower and upper index bounds for one dimension of the array and takes the form

  lower_bound :

  or

  :

  If the *lower_bound* is omitted it is taken to be 1.
- Example:

```
REAL FUNCTION assumed_shape(a, b)
   IMPLICIT NONE
   INTEGER, DIMENSION( : , : ) :: a
   REAL, DIMENSION( 5: , : , : ) :: b
   ...
END REAL FUNCTION assumed_shape
```

- The intrinsic functions SIZE, LBOUND and UBOUND have two arguments. The first is the name of the array. The second is optional; if present, it specifies the dimension; it must therefore be an integer, DIM, which lies in the range $1 \leq \mathsf{DIM} \leq$ *rank*.
  - If DIM is present, SIZE returns the extent of the specified dimension; if DIM is not present, SIZE returns the size of the whole array.
  - If DIM is present, then LBOUND returns the lower index bound of the specified dimension; if DIM is not present, the result of the function reference is a rank-one array containing *all* the lower index bounds
  - If DIM is present, then UBOUND returns the upper index bound of the specified dimension; if DIM is not present, the result of the function reference is a rank-one array containing *all* the upper index bounds.

- An **automatic array** is a special type of explicit-shape array which can only be declared in a procedure, which is not a dummy argument, and which has at least one index bound that is not constant. The space for the elements of an automatic array is created dynamically when the procedure is entered and is removed upon exit from the procedure. In between entry and exit, an automatic array may be used in the same manner as any other array – including passing it or its elements as actual arguments to other procedures.

  Example:

```
SUBROUTINE abc(x, y, n)
   IMPLICIT NONE
   !
   ! Dummy arguments
   INTEGER, INTENT(IN) :: n
   REAL, DIMENSION(n), INTENT(INOUT) :: x   ! Explicit-shape
   REAL, DIMENSION(:), INTENT(INOUT) :: y   ! Assumed-shape
   !
   ! Local variables
   REAL, DIMENSION(SIZE(y,1)) :: e          ! Automatic
   REAL, DIMENSION(n, n) :: f               ! Automatic
   REAL, DIMENSION(10) :: g                 ! Explicit-shape
   ...
END SUBROUTINE abc
```

- Automatic arrays are convenient when array space (of variable shape) is needed on a temporary basis inside a procedure, and such arrays are, therefore, often called **work arrays**.
- An **allocatable array** is an array whose rank is declared initially, but none of its extents, and which is subsequently allocated with bounds specified dynamically during execution. The space required for an allocatable array may be released at any time during execution by deallocating the array. Allocatable arrays cannot be dummy arguments, function results or components of a derived type. (. . .)
- In **whole array** processing two conformable arrays can appear as operands in an expression or an assignement, and the operation or assignement is carried out on an element-by-element basis. The following rules apply:
  - Two arrays are conformable if they have the same shape.
  - A scalar, including a constant, is conformable with any array.
  - All intrinsic operations are defined between conformable arrays.
- Example: If three rank-four arrays are declared as follows

```
REAL, DIMENSION(10, 10, 21, 21) :: x
REAL, DIMENSION(0:9, 0:9, -10:10, -10:10) :: y
REAL, DIMENSION(11:20, -9:0, 0:20, -20:0) :: z
```

  then the statement

```
x = y + z
```

has exactly the same effect as the following nest of DO loops

```
DO i = 1, 10
   DO j = 1, 10
      DO k = 1, 21
         DO l = 1, 21
            x(i,j,k,l) = y(i-1, j-1, k-11, l-11) + &
                         z(i+10, j-10, k-1, l-21)
         END DO
      END DO
   END DO
END DO
```

- The concept of function can be extended to **array-valued functions**. The following rules apply:
  - An array-valued-function must have an explicit interface.
  - The type of the function, and an appropriate dimension attribute, must appear within the body of the function, not as part of the FUNCTION statement.
  - The array that is the function result must be an explicit-shape array, although it may have variable extents in any of its dimensions.
- Many of Fortran's intrinsic procedures may be used in an **elemental** manner in whole-array expressions; in other words, they will accept arrays as actual arguments, and will return as their result an array of the same shape as the actual argument in which the procedure has been applied to every element of the array.
- Fortran provides three intrinsic functions specifically designed for vector and matrix operations, where it is assumed that matrices are stored in rank-two arrays and vectors are stored in rank-one arrays: MATMUL, DOT_PRODUCT and TRANSPOSE.
- Fortran 90 contains a large number of other intrinsic functions which operate on arrays of any dimension. Four of these, MAXVAL, MINVAL, PRODUCT and SUM, are also particularly useful for work with vectors and matrices.

Name	*Result*
MATMUL	Matrix product of two matrices, or a matrix and a vector
DOT_PRODUCT	Scalar (dot) product of two vectors
TRANSPOSE	Transpose of a matrix
MAXVAL	Maximum value of all the elements of an array, or of all the elements along a specified dimension of an array
MINVAL	Minimum value of all the elements of an array, or of all the elements along a specified dimension of an array
PRODUCT	Product of all the elements of an array, or of all the elements along a specified dimension of an array
SUM	Sum of all the elements of an array, or of all the elements along a specified dimension of an array

- **Masked array assignement** is a generalization of whole array assignement. It allows a finer degree of control over the assignement of one array to another, by use of a **mask** which determines whether the assignement of a particular element

should take place or, alternatively, which of two alternate values should be assigned to each element. It comes in two forms.

- The first, simpler, form is known as a WHERE **statement**, and takes the general form

  WHERE (*mask_expression*) *array_assignement_statement*

  where *mask_expression* is a logical expression of the same shape as the array variable being defined in the *array_assignement_statement* The effect is that the assignement statement is only executed for those elements where the elements in the corresponding positions of the *mask_expression* are true.

- Example:

```
REAL, DIMENSION(100) :: array
...
WHERE (array < 0.0) array = -array
```

  will change the sign of all the elements of array having negative values, and leave those having positive values unchanged.

- The second form of the masked array assignement is the WHERE **construct**, which takes the form

  WHERE (*mask_expression*)
  array_assignement_statement
  ELSEWHERE
  array_assignement_statement
  END WHERE

  or

  WHERE (*mask_expression*)
  array_assignement_statement
  END WHERE

  The effect of the WHERE construct is that the set of array assignement statements following the WHERE are only executed for those elements where the elements in the corresponding positions in the mask expression are *true*. Conversely, the set of array assignement statements immediately following the ELSEWHERE are only executed for those elements where the elements in the corresponding positions in the mask expression are *false*. Note that all the arrays being assigned values must be conformable with each other, and with the mask array.

- Example:

```
REAL, DIMENSION(100) :: arr
...
WHERE (array /= 0.0)
   array = 1.0/array
ELSEWHERE
   array = 1.0
END WHERE
```

  will replace every non-zero element of array by its reciprocal, and every zero element by 1.0.

- **Array sections** can be extracted from a parent array in a rectangular grid (that is, with regular spacing) using **subscript triplet** notation, or, in a completely general manner using **vector subscript** notation. In either case the resulting array section is itself an array, and can be used in the same way as an array.
- An array element has already been defined as

  array_name $(i_1, \ldots, i_k)$

  where *array_name* is the name of the array, k is the rank of *array_name*, and the i_j are subscripts. If any of the i_j are replaced by what are called subscript triplets or vector subscripts, then, instead of defining an array element, we have defined an **array section**. The **rank** of the array section so defined is the number of subscript triplets and vector subscripts it contains. An array element has rank zero.
- A **subscript triplet** takes the following form:

  subscript_1 : *subscript_2* : *stride*

  or one of the simpler forms

  subscript_1 : *subscript_2*
  subscript_1 :
  subscript_1 : : *stride*
  : *subscript_2*
  : *subscript_2* : *stride*
  : : *stride*
  :

  - *subscript_1, subscript_2* and *stride* are all scalar integer expressions.
  - A subscript triplet is interpreted as defining an ordered set of subscripts that start at *subscript_1*, that end on or before *subscript_2*, and have a separation of *stride* between consecutive subscripts.
  - The value of *stride* must not be zero.
  - If *subscript_1* is omitted, it defaults to the lower index bound for the dimension.
  - If *subscript_2* is omitted, it defaults to the upper index bound for the dimension.
  - If *stride* is omitted it defaults to the value 1.
  - The first colon must always be included, even if the first subscript is not specified.
- Example: If the array `arr` is declared as

  ```
  REAL, DIMENSION (3, 4) :: arr
  ```

  then

  ```
  arr(2, :)       is a rank-one real array whose elements are
                  arr(2, 1), arr(2, 2), arr(2, 3), arr(2, 4)

  arr(:, 3)       is a rank-one real array whose elements are
                  arr(1, 3), arr(2, 3), arr(3, 3)

  arr(1:2, 3:4)   is a rank-two real array whose elements are
                  arr(1, 3), arr(2, 3), arr(1, 4), arr(2, 4)
  ```

- A **vector subscript** is an integer array expression of rank 1, each of whose elements has the value of a subscript in the array section being defined.
- Example: If arr is a rank-one array of arbitrary size and type and v = (/ 3, 7, 4, 5 /) is a rank-one array of size 4 then the array section arr(v) is a rank-one array of size 4, whose elements are, in order, arr(3), arr(7), arr(4) and arr(5).
- Example: If the arrays p and u are declared as

```
LOGICAL, DIMENSION(3) :: p
INTEGER, DIMENSION(3) :: u = (/ 3, 2, 2, 3, 1 /)
```

  then p(u) is a rank-one logical array of size 5, whose elements are, in order, p(3), p(2), p(2), p(3) and p(1).
- Subscripts, subscript triplets and vector subscripts can be used together to define an array section.
- Example: If the arrays string and vec are declared as

```
CHARACTER(LEN = 10), DIMENSION(3, 4, 9) :: string
INTEGER, DIMENSION(5) :: vec = (/ 7, 1, 3, 1, 4 /)
```

  then string(vec, 3, 5:9:4) is a rank-two character array whose elements are

```
string(7, 3, 5)     string(7, 3, 9)
string(1, 3, 5)     string(1, 3, 9)
string(3, 3, 5)     string(3, 3, 9)
string(1, 3, 5)     string(1, 3, 9)
string(4, 3, 5)     string(4, 3, 9)
```

Fortran 90 syntax introduced in Chapter 13

Array declaration	*type*, DIMENSION(*dim_spec*, ...) :: *list of names* where each *dim_spec* (up to a maximum of 7) takes one of the forms: *extent* *lower_bound* : *upper_bound* *lower_bound* : : *upper_bound* :
Allocatable attribute	ALLOCATABLE
Allocate and deallocate statements	ALLOCATE (*list of array_specifications*, STAT = *stat_var*) ALLOCATE (*list of array_specifications*) DEALLOCATE (*list of allocated_arrays*, STAT = *stat_var*) DEALLOCATE (*list of allocated_arrays*)
Masked array assignment	WHERE (*conformable_log_expr*) *array_name* = *expression* WHERE (*conformable_log_expr*) *array assignement statements* END WHERE WHERE (*conformable_log_expr*) *array assignement statements* ELSEWHERE *array assignement statements* END WHERE
Array section	*array_name*(*subscript_triplet*) *array_name*(*vector_subscript*) where *subscript_triplet* is one of *subscript_1* : *subscript_2* : *stride* *subscript_1* : *subscript_2* *subscript_1* : *subscript_1* : : *stride* : *subscript_2* : *subscript_2* : *stride* : : *stride* : and *vector_subscript* is an integer array expression of rank-one

Example 13.1x

Problem (13.1x)

Write a program to illustrate the use of the intrinsic functions RESHAPE, TRANSPOSE and MATMUL.

Solution (13.1x)

```
PROGRAM vectors_and_matrices
   IMPLICIT NONE
   INTEGER, DIMENSION(2, 3) :: matrix_a = &
            RESHAPE( (/ 1, 2, 2, 3, 3, 4 /), (/ 2, 3 /) )
   ! matrix_a is the matrix       [ 1  2  3 ]
   !                              [ 2  3  4 ]
   INTEGER, DIMENSION(3, 2) :: matrix_b
   INTEGER, DIMENSION(2, 2) :: matrix_ab
   INTEGER, DIMENSION(2) :: vector_c = (/ 1, 2 /)
   INTEGER, DIMENSION(3) :: vector_bc
   !
   ! Set matrix_b as the transpose of matrix_a
   ! matrix_b = TRANSPOSE(matrix_a)
   ! matrix_b is now the matrix   [ 1  2 ]
   !                              [ 2  3 ]
   !                              [ 3  4 ]
   ! Calculate matrix products
   matrix_ab = MATMUL(matrix_a, matrix_b)
   ! matrix_ab is now the matrix  [ 14  20 ]
   !                              [ 20  29 ]
   vector_bc = MATMUL(matrix_b, vector_c)
   ! vector_bc is now the vector  [ 5  8  11 ]
   !
END PROGRAM vectors_and_matrices
```