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

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
```