An Introduction to FORTRAN
Course Notes[1]

Computer Services Centre
Australian National University

Last revision : September 1987

[1]These course notes are designed for use in conjunction with an introductory Fortran course taught by the Computer Services Centre, Australian National University.

FORTRAN (FORmula TRANslator), is a high-level computer language used in scientific and engineering applications. In 1966 a standard for FORTRAN was issued by the American National Standards Institute (ANSI). In 1977 ANSI issued a revised standard, comprising a *full* language specification and a *subset* language specification. This course teaches and refers to the 1977 Standard FORTRAN.

Consult the FORTRAN manual for your computer to see whether your version is the full, subset or a non-standard specification.

First printed : May 1976

Other revisions : September 1977
December 1977
November 1979
January 1981
March 1981
July 1981
August 1984
September 1987

Contents

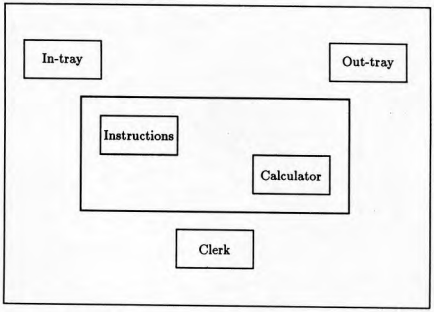
Chapter 1

Introduction

A computer is a tool used to solve certain classes of problems. For a computer to be able to solve a problem, we have to know how to solve the problem ourselves and then use the computer to apply the method to given data. The programmer specifies the method in a program; the computer provides speed and accuracy.

1.1 Automating a Problem Solution

The figure below shows a clerk seated at their desk with a calculating sheet in front of them. They have an *in-tray* into which slips are placed and an *out-tray* into which they place results. They also have a calculating machine to do the arithmetic.



The clerk's task is to take an input slip which contains the following information about employees

- name
- rate of pay

- number of hours worked

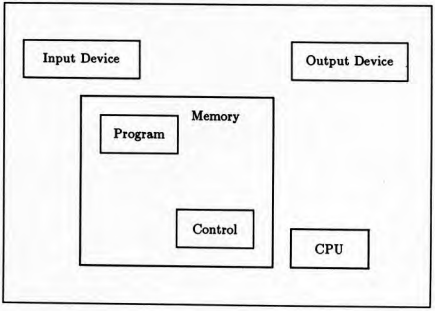
and they have to prepare an output slip with the above information plus calculations for

- overtime hours
- normal pay
- overtime pay
- total pay

A section of the calculating sheet contains instructions that enable them to perform the desired calculations. These instructions are

1. Take the next slip from the in-tray and copy the name, rate and total hours.
2. Calculate the overtime hours as (Total hours – 40).
3. Calculate the normal earnings as (40 × Rate).
4. Calculate overtime earnings as (Overtime hours × Rate × 1.5)
5. Calculate gross earnings as (Normal earnings + Overtime earnings).
6. Prepare an output slip and put it in the out-tray.
7. If there are any more slips in the in-tray go to step 1, otherwise stop.

Now let us automate this system with a computer



The in-tray is replaced by an *input device*. There are many types of input devices that may be attached to a computer. We will consider it as a convenient means of putting information into the machine. The out-tray is replaced by an *output device*. There are many types of output devices that may be attached to a computer. We

will consider it as a convenient means of getting information out of the machine. The instructions are coded and stored in a *program*. The calculating sheet, with both its instruction area and work area, is replaced by the *memory*. Memory is a temporary storage area which stores the program and its calculations during execution of the program. The calculator is replaced by the *Central Processing Unit (CPU)*. This performs the calculations. Finally, the clerk is replaced by a *control* program. The function of the control program is to take each instruction in turn from the memory and cause the appropriate action to take place. This might be

1. To read some information from the outside world via the input device.
2. To write some information to the outside world via the output device.
3. To calculate some quantity, using the central processing unit.

1.2 Solving Problems Using Computers

The steps involved in solving problems by using computers are

1. Formulate the problem carefully and determine exactly the objective to be reached.
2. Find a method for solving the problem. The method is referred to as an *algorithm*.
3. Organise the information associated with the problem in a way suitable for processing by the computer. This is called preparing the input data.
4. Prepare instructions for the computer. A program is a sequence of instructions, which is a translation of the algorithm found in step 2 into a computer language.
5. Run the program on the computer, with the input data, to produce an answer to the problem.

If you wanted to solve a problem without using a computer, you would still carry out the first two steps, and some modification of the third.

1.3 Programming Languages

Each computer has its own specific language called its *machine language*, or *assembler*. Programming in machine language is very tedious because

- Detailed knowledge of how the computer works is necessary.
- The program will only be able to run on the type of machine for which it was originally written.
- The machine instructions are very primitive, and very many of them are required for a simple program.

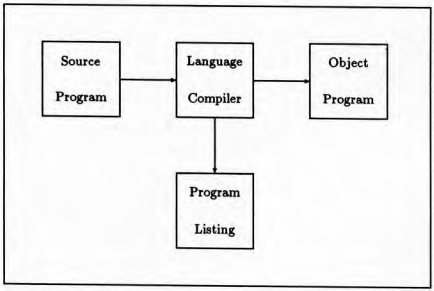
Attempts to overcome such problems have resulted in the so-called *high level languages* such as FORTRAN, ALGOL, PASCAL, PL/1, COBOL and SIMULA. A computer cannot directly understand a high level language, so a program called a *compiler* is used to read the high level language and translate it to machine language. Steps involved in using a high level language are

1. Analyze the problem and develop an algorithm to be used to solve the problem.
2. Translate the algorithm to a program written in the high level language. Enter the program into the machine, usually by typing it in at a terminal, and store it permanently, usually in a file on a disk. This is referred to as the *source program*.
3. Compile the source program into machine language (called the *object program*) using a compiler. This is referred to as compilation of a program, and this occurs at *compile time*.
4. Collect all object programs to form an *executable program*. This is also called *linking*.
5. Execute the program, if necessary using data (which is also usually stored in the machine). This occurs at *execution time*.

The phases are represented below. It is important to understand the different phases that a program will go through before it is actually executed.

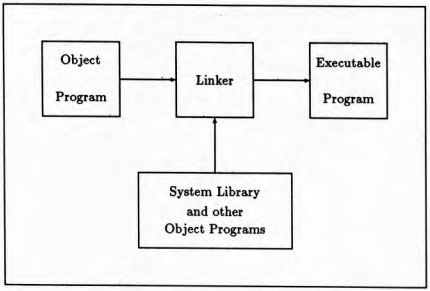
1.3.1 Compile Time

The compilation (or translation phase) that your program goes through may be represented by the following diagram.



1.3.2 Linking Time

The output from the compile phase, the object program, is the input for the next phase, that of collecting together all the object programs that are necessary to form an executable program, which is what the computer will execute. Object programs come from your program, the system library, and other subprograms that may have been written by other people or by yourself.



1.3.3 Execution Time

Execution time refers to the execution of your program. Note that much has happened to your program prior to it reaching this point on its way to producing results for you.

1.4 Operating Systems

An *operating system* is a large complex program used to control the operation of a computer. All programs execute under the guidance of the operating system. Part of the operating system functions are

- General job scheduling.
- Identify each user and label each job's output.
- Check account numbers and passwords.
- Monitor jobs to ensure they do not exceed a maximum time.
- Load the required compilers into the computer as needed.
- Control activities of input and output devices.

The programmer must provide special *control statements* for the operating system which will specify such things as

- The account number and password, to log on.
- Which compiler to use.
- The location of the program and data.
- Where to send printed output.

Such control statements must be provided with each program that is to be run, and they must be prepared according to a fixed format that is dependent upon the particular computer installation. The operating system control statements are independent of the FORTRAN language, but must be used for each run.

Chapter 2

Variables, Expressions and Statements

2.1 Computer Memory

The main memory of a computer consists of entities called *words*. Each word can store a certain amount of information. The amount of information in a memory word varies according to the computer. Computer memory is vastly different to human memory. Humans tend to display the use of their memory by saying such things as 'I remember when I was last in Thargomindah and all the frogs climbed trees'. With a computer, if you put something into memory, then it will stay there until you put something else in its place. An analogy to this is recording a piece of music on a cassette tape. The cassette tape remembers what you recorded on it, to the extent that it will play it back, and this recall will continue until you record something else on it. Another analogy is if you had a box and you placed a blue sheet of paper in it. If you looked inside this box some time later and saw that the blue sheet was still there, would you say that the box had remembered the sheet of paper? Probably not, but this is how a memory word in the computer 'remembers'.

2.2 What is Stored in Computer Memory

During your program's execution, your program and its calculations are stored in some part of memory. After the execution finishes, that part of memory is re-used for other programs. Some of the things that are stored in the computer memory are

Integer numbers, e.g. 123, –724 — There is a limit on the size of an integer which varies from computer to computer.

Real (or floating point) numbers, e.g. 62.43, –0.74 — A real number is stored as two parts, the mantissa and the exponent, e.g.

$$7.3 \times 1000000$$

The magnitude (sign not considered) of a real number must be zero or lie between limits that vary between computers. Only a certain number (machine dependent) of significant digits is stored in memory. If a real constant is

specified with too many significant digits, then the excess is just dropped (that is, the value recorded is truncated) without issuing a diagnostic. (Larger or smaller real numbers with more significant digits may be used in FORTRAN by declaring them in a special way.)

Computer instructions — This is done automatically for us by the FORTRAN compiler and so the exact form does not concern us.

Text or messages, e.g. FRED — The computer stores characters by using a digital code for each possible alphabetic (A,B,C,...,Z), numeric (0,1,2,...,9), or special ($,/,*,space) character. Fortran stores characters in a code called ASCII, or EBCDIC depending upon the computer used.

These four different types (and others) are stored as some combination of zeros and ones (i.e. a *binary* code) in a computer word. The representations in each case are different, so the real number 3.0 will be stored in a completely different way to the integer 3. Suppose a word contains an integer number. If this word is then referenced as a real number, we should not expect it to be the correct value. It is therefore important that these different types be used correctly. Memory stores all information unselectively, and information only has meaning when you reference it as a certain type.

2.3 Names

Fortran allows us to reference locations in the computer memory by the use of names. Any reference to this *name* will reference the corresponding *location* in the computer memory. The information stored in a location may be of two types. A *constant* references a location in memory whose value remains fixed for the duration of execution of the program. A *variable* refers to a memory location whose stored value may be changed during the execution of the program. An analogy to this is in a bank vault. A bank vault contains deposit boxes in the names of the bank's customers. The boxes are distinguished by a name (being the customer name and account number). So we can refer to box name `MACINTOSH551549` and put something in the box. Note the difference between the name of the box and its contents. This kind of box is analogous to the variable in FORTRAN as its contents can be examined or changed.

2.3.1 Rules for the Formation of Constant Names

The word containing a constant number is given a symbolic name which is the same as its value. Unless a minus sign precedes a constant, it is assumed to be positive.

- An integer constant is written as a signed or unsigned string of digits without a decimal point, e.g.

```
1, -10, 31234
```

- A real constant is written as a signed or unsigned string of digits containing a decimal point, e.g.

```
10.23, 6., -72.189
```

Very large or small numbers may be written with an integral decimal exponent. For example,

`128700000.0` is equivalent to `128.7E6`

`0.0000125` may be written as `0.125E-4`

Other examples are `-0.179E24`, `1278.0E-10`

Note that the number to the left of the `E` may be an integer, but the whole name is the name of a real quantity. For example

`42E3` is the same as the real number `42000.0`

- A literal constant is written as a string of characters enclosed by single quote characters, e.g. `'FRED'`, `'I AM'`

The major difference between an integer and real constant is the absence or presence respectively, of a decimal point (or the use of the `E` format). Thus `3` is an integer constant, but `3.0` is a real constant. The forms are not interchangeable as they are stored and processed within the computer in entirely different ways. Note that `3.0`, `3.`, and `3.00000` are all equivalent.

2.3.2 Rules for the Formation of Variable Names

A variable name consists of one to six alphanumeric (i.e. alphabetic and numeric) characters, the first of which must be alphabetic. By default, if the name starts with one of the letters I,J,K,L,M,N, then the variable is of type *integer* and refers to a word that may store integer numbers only. If the name begins with one of the other alphabetic characters, A to H, O to Z inclusive, then the variable is of type *real* and refers to a word that may contain real numbers only. For example,

`I, I10, NAME`	are integer variable names, while
`A, DATA, PLANE`	are real variable names.

It should be noted that the compiler places no significance on variable names beyond inspecting the first letter to establish whether the variable is integer or real. A name such as `B7` does not mean `B` times `7`, or `B` raised to the 7th power.

Good programmers assign variable names that make it easy to remember the meaning of the variable, but no such meaning is attached by the Fortran system. It should also be noted that every combination of letters and digits is a separate name, although lower case letters are not distinguished from upper case. Thus `A`, `AB`, `AB8` and `ABC` are all different and `ABC` is the same as `Abc` or `abc` but not the same as `BAC`.

2.4 Arithmetic Expressions

Arithmetic expressions may contain

- variable names,
- constants,

- arithmetic operators, and
- brackets.

The arithmetic operators are

+	representing addition,
-	subtraction,
*	multiplication,
/	division, and
**	exponentiation.

2.4.1 Formation of Arithmetic Expressions

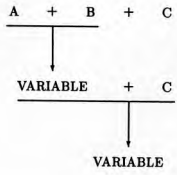
The basic expression is a single operation and has the form

<variable or constant> <operator> <variable or constant>
or

<operator> <variable or constant>

Examples:- `A*2`, `A-B` and `-B` are basic expressions.

The result of evaluating an expression is a variable or constant. Complex expressions are built up from basic expressions, with operations being done one at a time. For example, `A + B + C` is evaluated in two stages as follows



Variables and constants must be separated from other variables and constants by arithmetic operators. Two arithmetic operators cannot be adjacent. If this situation arises, then the operators must be separated by brackets. For example, `A*-B` is illegal and should be `A*(-B)`. The error could also be corrected by using `-B*A`.

2.4.2 Order of Evaluation of an Arithmetic Expression

Arithmetic expressions in algebra may be interpreted in different ways. This ambiguity is not acceptable in a programming language, so there are strict rules which determine the order of evaluation of an otherwise ambiguous expression. For example, consider the expression

```
HOURS*RATE + BONUS
```

Do we do the multiplication first, or the addition first? There is a vast difference in the answer we may get. In evaluating an expression, the order is

1. brackets (innermost first),
2. exponentiation,
3. multiplication and division,
4. addition and subtraction.

For example, the expression `A*B-C/D` is evaluated as if it were written `(A*B)-(C/D)`. Where two operators have the same priority (e.g. multiplication and division), the order is taken from left to right. For example, the expression `A/B*C` is evaluated as `(A/B)*C` which will probably be different from `A/(B*C)`.

Exponentiation is the exception : `I**J**K` is evaluated as `I**(J**K)` and not `(I**J)**K`.

A more complex expression is

```
A + ((B-C)*D + (E-F**K))/3.2*G+H
```

This is evaluated as follows

1. The expression `((B-C)*D + (E-F**K))` is in brackets so it will be evaluated first. Within this expression
   (a) `(B-C)` is the leftmost bracketed expression, so it will be done first.
   (b) `(E-F**K)` is the next bracketed expression to be evaluated. Within this expression, `F**K` is evaluated then `E - (F**K)` is evaluated
   (c) Step 1a `* D` is evaluated.

   Finally the result for step 1 is obtained by evaluating steps 1c + 1b
2. The result from step 1 is divided by the real constant `3.2`
3. The result from step 2 is multiplied by `G`
4. `A` is added to the result from step 3
5. `H` is added to the result from step 4 giving the final answer.

Another way of representing the above explanation is to place a number under the arithmetic operators to indicate the order in which they are applied.

```
A  +  ((B  -  C)  *  D  +  (E  -  F  **  K)) / 3.2  *  G  +  H
8        1     4     5        3     2     6     7     9
```

A quick method of checking that an expression has the correct number of right and left brackets is to count each bracket, adding 1 to the total for each left bracket and subtracting 1 for each right bracket. The total should be 0. In the above example

```
A + ((B - C) * D + (E - F ** K)) / 3.2 * G + H
    12     1       2          10                  correct!
```

In the erroneous expression

```
A + ((B - C) * D + (E - F ** K) / 3.2 * G + H
   12     1       2          1
```

a positive total indicates a surplus of left brackets but does not show where the error is.

2.4.3 Expression Type

Integer expressions and real expressions are arithmetic expressions whose resulting values are of type integer and real respectively. The result of an expression containing one operand, such as `-B`, has the same type as the operand. The result of a basic expression with two operands, such as `A + B`, has the same 'type' as its components if they are both the same type. If both components are integer then the resultant variable or constant will be integer. For example

`I + J` gives an integer result.
`X/Y` gives a real result.

2.4.4 Integer Division

This is the division of an integer quantity (either a constant or an integer variable) by another integer quantity. The result of the division is an integer. Where the division is not exact, the result is *truncated* to an integer value. That is, the result of `12/5` is `2` and not `2.4`, while the result of `-9/4` is `-2` and not `-2.25` . As a result, `10/3*4` is `12` while `10*4/3` is `13`. Also, `10/3*3` is `9` and not `10`. **N.B.** It is *not* rounded. Note that `2**(-3)` is equivalent to `1/(2**3)`, which is `0` and not `0.125` .

2.4.5 Mixed Type Expressions

FORTRAN allows mixed type arithmetic expressions. A mixed type arithmetic expression is one that contains more than one type of variable or constant (i.e. a mixture of integer and real). If there is a mixture of real and integer, then the integer is converted to real, internally by FORTRAN, and the result is of type real.

The result of `10/3 + 4.2` is `7.2`
The result of `10/3.0 + 4.2` is `7.533`

2.5 The Assignment Statement

An assignment statement is of the form

variable = expression

where variable is a legal variable name, and
expression is an arithmetic expression.

The assignment statement causes the expression on the right to be evaluated and the resulting value to be stored in the variable on the left hand side of the = character. The assignment statement is read as

variable BECOMES the value of the expression

as the variable's value will become the value of the expression, thus losing any previous value that it may have had. The expression on the right is evaluated according to the rules we have seen for evaluating expressions. References to variables on the *right* cause their values to be fetched from memory. It is impossible to fetch a value of a variable from memory unless it has already been given a value somehow, prior to this point in the program. The expression on the right is said to be *evaluated.* The name on the *left* is the place in the computer memory in which the result of the evaluation is stored. For example, after the statements

```
A = 3.0
B = 12.3
C = -10.2
D = 47.1
D = A*B+C
```

are executed, the values of the variables will be

```
   A        B        C        D
 3.0      12.3     -10.2     26.7
```

We can now say such things as:

`3.0` has been stored in `A`
`12.3` is the current value of `B`
`C` is assigned the value `-10.2`
`D` is `26.7`

The statement `I = I + 1` is legal and results in the value of the variable `I` being incremented by 1. The `=` character is not an *equals* sign in the sense of algebra, because it does not test for equality. It causes the replacement of the current value of the variable on the left hand side with the result of the expression on the right hand side.

2.5.1 Type Conversion Across the Assignment Operator

In the statement

variable = expression

the type of the variable (i.e. integer or real) and the expression do not necessarily have to be the same. For example,

```
(a)     I = A + B
(b)     A = I * J
```

are both legal. The expression is evaluated according to its type (real for (a) and integer for (b)). The result is then converted to the type of the variable on the left hand side before being stored. So the statement `A = 3` results in `3.0` being stored in `A`. The statement `ICE = 4.8` results in the number `4` being stored in the variable `ICE`.

2.6 Overflow, Underflow, and Divide Check Errors

If a result attains a magnitude which is greater than the largest (positive or negative) value that can be stored, then an *overflow* is said to have occurred. An *underflow* occurs when the magnitude of a real result is less than the smallest value allowed. Division by zero (either integer or real) is not defined and results in a *divide check* condition. All of the above are considered to be errors, but underflow is generally considered far less serious. These errors may be handled differently on different machines, and you should acquaint yourself with the actions taken for these errors on the computer you are using. The actions may vary from ignoring the error (and not telling you), or reporting that it happened somewhere in the program, to stopping the program when one occurs.

2.7 Exercises

Exercise 2A

Write FORTRAN statements to evaluate the following algebraic formulae.

1.

$$z = \frac{a+b}{c+d}$$

2.

$$z = a + \frac{b}{c+d}$$

3.

$$z = \frac{a+b}{c} + d$$

4.

$$z = a + \frac{b}{c + \frac{d}{e}}$$

5.

$$z = \frac{n(n-1)}{2}$$

6.

$$z = \frac{(a-b)(c-d)}{e(f+g)}$$

7.

$$y = -2.314 + (5.67z - 3.29 \times 10^{-4})z^3 + 4.13z^7$$

Exercise 2B

Identify each of the following as either a *real* constant, an *integer* constant, or neither.

1. 0.001
2. .2
3. 77
4. 1-23
5. 87E-05
6. 3435.11
7. $66
8. 6.1E-.5
9. 467+1
10. 43.1.2
11. 223.

Exercise 2C

Identify which of the following are legal variable names. Which ones are *integer* variable names, and which are *real* variable names?

1. QAZ
2. COMPUTER
3. REAL
4. 69
5. CD-5
6. L9A5Z2
7. TUFF.
8. BSCBSC
9. A+B
10. 3XY
11. AJKLMN
12. BLOOD

Exercise 2D

What is the error in each of the following Fortran arithmetic statements? Notice that although we may recognise errors in the statements below, we cannot correct them, as there are many possible changes that may be made, which would correct them. The Fortran compiler will also adopt this approach, and these statements would result in errors occurring at compile time.

1. `X = A + 3B`
2. `Y = ((A+B)*(C+D)-(A-D)*B-C))`
3. `3.14159 = PI`
4. `J = M-4.5*N**-2`
5. `AMOUNT = BALANCE + RECEIPT - SALES`
6. `X+Y = (A+B)/Z-2*PI*R`
7. `TOAST = (TOAST + BUTTER)(BREAD/TOAST)`

Exercise 2E

What would be the result of executing the following statements? The initial values of the variables are

```
A = 6.0      X = -9.0      J = 2
B = 3.6      I = -3        N = 5
```

1. `Z = (A+B)/X`
2. `Z = J+I+N`
3. `Z = (J-I)/N`
4. `Z = N-I/J`
5. `Z = N/J+I+2`
6. `K = B`
7. `K = N/J`
8. `K = B*3.0 + X`
9. `K = (A+B-X)/3.0`
10. `K = I*J/N`
11. `K = I/N*J`

Chapter 3

Program Layout and Input/Output

3.1 The Fortran Character Set

A Fortran program is written using the following characters

1. The twenty-six upper case alphabetic characters `A` through `Z`. (Lower case letters are converted to upper case internally by Fortran.)
2. The ten numeric digits `0` through `9`.
3. The thirteen special characters

	blank
=	equals
+	plus
-	minus
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point
$	currency symbol
'	apostrophe
:	colon

The full Ascii or Ebcdic (depending on the machine type) character set is legal in literal constants, e.g. 'why?', '[...]'.

3.2 Spaces in Fortran

Except for certain specified uses, e.g. as part of a literal constant, spaces or blanks have no meaning and may be used freely to improve the appearance of the program. For example,

```
AMP = AJ + 3.6
```

and

```
A    MP  = AJ+ 3.   6
```

and

```
AMP=AJ+3.6
```

are equivalent, the first being the most desirable.

3.3 Fortran Program Layout

A program is a sequence of statements and comments written on 80-column lines. Each statement of a program begins on a new line. The 80 columns are divided into a number of fields with different uses. Only the first 72 columns are read by the computer.

3.3.1 Statement Line

A statement is written in columns 7 through 72 . The only Fortran statement covered so far is an assignment statement. So an example of a statement is shown below (note: blanks are shown as ␣).

␣␣␣␣␣␣AMA␣=␣(BAG-12.6)␣+␣2345.2*FEE␣-␣DOC

Note how there are six blanks at the start of the line, so that the statement starts in column 7. For good program layout you should start most Fortran commands in column 7 (rather than after column 7). This makes your program easier to understand, easier to read, fix and alter. Later we will learn when it is appropriate to indent past column 7.

3.3.2 Continuation Line

Often a statement may require more than columns 7 through 72 and so a method of continuing the statement is required. This may be done by putting a non-zero, non-blank character in column 6 of the line that is the continuation of the previous line. Up to 19 continuation lines are allowed for a single statement. An example of this is if we wanted to spread the following statement over 3 lines

␣␣␣␣␣␣INCHES␣=␣MILES*1760*3*12␣+␣YARDS*3*12␣+␣FEET*12␣+␣INS

Although this is not normally done if it is not necessary, the following three lines have an identical effect to the one line above.

```
␣␣␣␣␣␣INCHES␣=␣MILES*1760*3*12
␣␣␣␣␣$␣␣␣␣␣␣␣␣␣+␣YARDS*3*12
␣␣␣␣␣$␣␣␣␣␣␣␣␣␣+␣FEET*12␣+␣INS
```

When using continuation lines, it is best to indent the continued lines to highlight the continuation, as in the above example. This practice makes your program easier to read and understand.

3.3.3 Comment Line

The letter C (or an asterisk) in column 1 designates that line as a comment, and whatever follows on that line is the text of the comment. A comment line does not affect the program in any way, and is available as a means of documentation. A line containing only blanks in columns 1 to 72 is also treated as a comment, and should be used to space out the program. Comments are allowed between the lines of a statement which has continuations. If a line has a C in column 1 the rest of the line is *ignored* by Fortran, but it will be printed in the listing of the program. Comment lines may not be continued. To have a multi-line comment, put a C in column 1 of each additional line. All programs should commence with a series of comments to explain the following details

1. The name of the author of the program.
2. The date
3. User documentation : What the program does from the point of view of a user of the program, including information as to how the user should prepare any required input data for the program.
4. Technical documentation : What the program does from the point of view of a person trying to understand the Fortran program.
5. Any limitations of the program, and how they may be overcome — e.g. a tendency to overflow.

The comments at the program head should be made using English sentences rather than pseudo Fortran sentences. It is easier to locate the different comment sections if standard headings are used and if the comments themselves are indented (as shown below). For example, you may find the head of a program has

```
C
C     Author:  Fortesque Quincy Zladinov Mc Smith
C
C       Date:  27th June 1976
C  Modified:   January 1981  by Les Landau
C              Modifications were to clarify comment
C              headings
C
C  Program Description:
C              This program finds the average of numbers
C              read in from input data. Any number of
C              data lines may be read. The last line
C              contains a sentinel (the number -999) and
C              so no other line may contain this value.
C              To change the ending indicator, change
C              the value of IFIN in the program.
C
C  Input Description:
C              The numbers to be read in must be integers,
C              one per line right justified in columns
C              1 - 6. The last line to be read in is a
C              line with -999 in it. This special ending
```

```
C            indicator may be altered by changing the
C            value of IFIN in the program below.
C
C   List of Variables Used:
C
C   AV.......CONTAINS THE AVERAGE OF THE NUMBERS
C   IFIN.....CONTAINS THE ENDING INDICATOR
C   N........CONTAINS THE NUMBER JUST READ IN
C   NUM......CONTAINS THE NUMBER OF DATA LINES READ
C   TOT......CONTAINS THE TOTAL OF THE NUMBERS
C
C
```

The above may seem to be rather verbose and unnecessary, however it does explain all that a person needs to know to either *use* or *modify* the program. Remember that incorrect or misleading comments are far worse than none, so ensure that those that you use are accurate! In addition to the above program header, there should also be comments within the program, describing each logical section. Appendix 1 contains information regarding the commenting of programs in general and specifically on what is expected for your assignments. You should read this appendix before attempting your assignments.

3.3.4 Statement Label

Optionally, a statement other than a comment may be labelled so that it may be referenced in other statements. In Fortran, a statement should not have a label unless it is referenced by some other statement in the program (we will see such commands as `DO` and `GO TO` later). A statement label contains from one to five digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the first line of a statement, i.e. not on continuation lines. Leading zeros are not significant, nor are blanks. The same statement label may not be given to more than one statement in a program, otherwise the label is not unique and a reference from another statement to this label would be ambiguous. A statement label is also known as a statement number. Although it is not mandatory, statement numbers are generally allocated in ascending order so that any statement number may be easily found without having to search the complete program. Statement numbers are usually multiples of 10 so that new statement numbers can be easily inserted later. For example,

```
60␣␣␣␣RAIN␣=␣PERIOD*MOIST␣+␣0.1*PROB
```

3.3.5 Sequencing

Columns 73 through 80 are ignored by the Fortran compiler but are printed as part of the compilation listing. These columns may be used to contain sequence numbers, so that if the program is on cards and the card deck is dropped, it may be easily put back in order by sorting on the sequence numbers. As cards are very seldom used nowadays, sequence numbers are not often used. The program name may be put in the first four columns of this field, with numbers in the next four columns. As in the case of statement labels, the numbers may be incremented by 5

or 10 to allow for lines that may need to be inserted at a later stage. Programs for this course do not require sequencing.

3.4 Structure of a Fortran Program

A program consists of a number of lines of Fortran statements. The program is executed sequentially, starting at the first executable statement. A comment is not executable. The execution of each statement is completed before going on to the next one.

3.4.1 End Statement

The physically last statement in a program must be an `END` statement and must be placed anywhere in columns 7 through 72. This statement indicates to the compiler that this is the last statement to be compiled.

3.4.2 Stop Statement

When a `STOP` statement is encountered, the execution of the program terminates. There must be at least one `STOP` command in a Fortran program. Remember the difference between compile time and execution time, as covered in Chapter 1? Understanding this difference will help in understanding the different functions of the `END` and `STOP` statements.

3.4.3 Simple Fortran Program

So a valid Fortran program is

```
C     This is a sample program
C
      a = 2.0
      b = 3.0 / a
      STOP
      END
```

Each of these statements is entered on a separate line. A program may have several `STOP` commands, but only one `END` directive. Note that there is no output from this program, and that values of variables are not kept after the program stops.

3.5 Reading and Writing in Fortran

Programs must converse with the outside world, to locate any data to operate on, and to display or save the results of any calculation. In batch mode this is done with data files for input and a log file or other files for output. When running interactively you can read and write to a terminal that is executing the program, and also read and write to data files. Fortran statements to do this are known as *Input/Output* or *I/O* statements. Initially we will use free format (sometimes called list directed) input and output, and only refer to the terminal for input and the terminal for output. Later we will discuss fixed format I/O and reading and writing data files.

3.5.4 Write Statement

The **WRITE** statement is very similar to the **READ** statement and has the form

```
WRITE (6,*) output list
```

where 'output list' is a list of variable names or expressions, separated by commas, whose values are to be written out by the program. Analogous to the **READ** statement, the list indicates the variable names whose values are to be printed. Unit 6 means that output is to be written to the terminal screen. Up to 132 columns per line are written. For example, to print the numbers that were read in the first example, the program could be

```
READ (5,*)  weight, age, money
WRITE (6,*) weight, age, money
STOP
END
```

Try this program on the computer so that you may see what happens. Don't forget to enter a line of data with the three numbers on it.

3.5.5 Examples of Read and Write Statements

READ statements should be used to read data that may vary from one run of the program to another. Consider the following problems

1. Find the average of the ten numbers 1.3, 2.4, 5.4, 6.3, −3.7, 0.0, 13.4, −12.0, 17.7, −21.3 . In this problem, all data has already been defined and so there is no need to read any data. The program would be

```
C     Find the average of ten specified numbers
C
      fnum = 10.0
      aver = (1.3 + 2.4 + 5.4 + 6.3 - 3.7 + 0.0 +
     $        13.4 - 12.0 + 17.7 -21.3)/fnum
      WRITE (6,*) aver
      STOP
      END
```

2. Find the average of any ten numbers. In this problem, the values of the ten numbers are not known when the program is written. Thus, they must be supplied as data and so must be read by the program which could be as follows

```
C     Find the average of any ten numbers
C
      fnum = 10.0
      READ (5,*) val1,val2,val3,val4,val5,val6,val7,
     $           val8,val9,val10
      aver = (val1+val2+val3+val4+val5+val6+val7+
     $      val8+val9+val10)/fnum
      WRITE (6,*) aver
      STOP
      END
```

Note that the program can only read real numbers. The program would have to be altered to read integers.

3. Find the average of any number of numbers. Now, not only the values of the numbers, but also the number of numbers, are not known when the program is written. The program would thus have to read a value indicating how many values there were, and then read that many values. It is not yet possible to write this program, but it will be set as an exercise later.

3.6 Writing Out Heading Information

The method below of writing out headings is not standard, but because it is easy we will use it initially, and will discuss the more standard method later. The output of numbers on their own is quite often confusing to interpret. Tables of figures in books and even single results usually have some accompanying text to indicate what the results mean, or how they were obtained. Fortran prints headings on the output with **WRITE** statements. To write a heading that will appear at the beginning of a list of values that are to be written on a line, include the heading in the output list, enclosed in single quotes. If you want to include a quote symbol within the heading itself, then the quote must be followed immediately by another quote. For example, to write out values for variables **eccles** and **jim** after a heading of **Goon with the wand**, the statement is

```
WRITE(6,*) 'Goon with the wand', eccles, jim
```

Headings may be written out on their own, without any variable values. In this case the corresponding **WRITE** statement simply has the heading on its own. For example, to write out

```
Min, Min, mighty modern Min
Henry, what have you done with the paper?
Ah Min, it's in the box marked       3
```

the following lines of Fortran may be used

```
i = 3
WRITE (6,*) 'Min, Min, mighty modern Min'
WRITE (6,*) 'Henry, what have you done with ',
$           'the paper?'
WRITE (6,*) 'Ah Min, it''s in the box marked', i
```

3.7 Program Debugging and Diagnostics

The source program is entered into the computer and becomes input to the Fortran compiler. If the compiler detects any errors it writes messages describing the errors. These messages are called *diagnostics*. Errors in a program are often referred to as *bugs* and the process of submitting a program for compilation, examining diagnostics, resubmitting the changed program, and repeating this, is called *debugging* a program. The compiler will detect most errors in the *syntax* or grammar of the Fortran program. The *semantics* or meaning of the program may still be 'wrong'. The answers may be wrong in that they may not be the desired ones, but the fault is not in the computer or the compiler. In this case the fault is with the author of the program, who either has a faulty algorithm or has failed to correctly transform the algorithm into a program or has faulty input data.

3.7.1 Debugging Tools

Intermediate Output — A program is normally written to solve a problem and output an answer. If a program is producing incorrect answers, the errors must be located and fixed. In theory all errors can be located by reading the program and examining the input data, but many errors are difficult to locate this way, due to their subtlety and the complexity of some programs. Sometimes it is difficult to know in which part of a program an error has occurred — indeed, even to know which parts of the program have been executed, and in what order. Intermediate `WRITE` statements within the program enable you to easily follow what is actually happening — this may then be compared to what should be happening and so errors may be localised, detected and corrected. The `WRITE` statements may then be removed from the program. If the value of the answer is incorrect then the programmer may request intermediate output of the values of some variables, at key points in the program such that the programmer has some idea of what their value should be. Then he can, hopefully, detect where the incorrect values first occur and so detect, or at least narrow the search for the location of the error.

Simple Test Data — Choose simple input data for which there are known answers and see that the program will produce the desired results. The design of test data is very difficult and is usually not exhaustive. The test data should cover extremes as well as typical data.

Desk Checking — This is a hand calculation performed by the programmer going through the program step by step. At each step the necessary calculations are performed, and the values of all the variables are recorded. By 'each step' it is meant each instruction or part instruction in the program. In a sense the programmer is to 'play the role of the computer'. The objective is not to check the arithmetic of the computer, but to force the programmer to focus attention on each detail of the program.

Manufacturer-supplied Debugging Aids — Some Fortrans come supplied with a debugging package. There are two types of debugging aids available — interactive and static. These aids enable you to trace through the program, print out values of variables when they change, do subscript checking and many other things.

3.8 Exercises

Exercise 3A

Write a program to do the following — Read in 4 numbers. The first two should be real, the next two should be integer. Write out the numbers in reverse order.

Exercise 3B

Write a program to do the following — Read two integers i and j. Do the following calculations and write out the results.

$$k = i + j$$

$$l = i \times j$$

$$m = \frac{i}{j}$$

$$n = i^j$$

Exercise 3C

Write a program to do the following — Read a positive real number. Place the integral part (ie, the whole number part) into a variable called `INT` and the fractional part into a variable called `FRACT`. Write out the number and `INT` and `FRACT`. For example, if the number is 53.261, then `INT` = 53 and `FRACT` = 0.261

Exercise 3D

Write a program to do the following — Read in two real numbers. Calculate the difference between the first one squared and the second one squared. Write out the the two numbers, their squares and the difference.

Chapter 4

The If Statement

4.1 Relational Expressions

A *Relational Expression* compares the values of two arithmetic expressions. The expressions are separated by a *relational operator* and the result will have the value *true* or *false*, corresponding to the result or condition of the relation. The relational operators are

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

Some examples of relational expressions are

1. `pay .LT. credit`

   The arithmetic expressions involved here are simply single variables. The relational expression will have the value `TRUE` if the value of the real number stored in `pay` is less than the real value stored in `credit`. Otherwise (i.e., if `pay` is greater than or equal to `credit`), it will have the value `FALSE`.

2. `nurke*2 .GE. min**3/ko`

   Here the arithmetic expressions are a little more involved. The relational expression will have one of the following values. `TRUE` if `nurke*2` is greater than or equal to `min**3/ko` or `FALSE` if `nurke*2` is less than `min**3/ko`

3. Brackets may be used within the arithmetic expressions involved, as in

   ```
   (kold*ice) / (man-kg)**4 + 7 .EQ. ((mine - iodine)*3)**jewel
   ```

Relational operators have a lower order of precedence than arithmetic operators, so the arithmetic expressions are evaluated first.

4.1.1 Use of .EQ. With Real Variables and Constants

The computer cannot represent most real numbers exactly due to the nature of storage of these numbers in a fixed size memory word. For example, in decimal

arithmetic, $\frac{1}{3}$ cannot be represented exactly in 6 significant digits, and is approximated by 0.333333. If this is then multiplied by 3, then you have (approximately) 0.999999 which is not exactly equal to 1.0 . Also, errors may be introduced when performing arithmetic operations on real numbers. For these reasons, it may not be meaningful to form a relational expression using `.EQ.` between two real arithmetic expressions. For example, 0.1 cannot be represented exactly in a computer word and in fact could be `0.1 + e` where `e` is a very small error. If this is then multiplied by 10, the result is

$$10 \times (0.1 + e) = 1.0 + 10 \times e$$

which has ten times `e`. Hence, 10 x 0.1 is not *exactly* equal to 1.0

4.2 Logical Expressions

A logical expression is a relational expression, or a combination of relational expressions. These expressions may be combined with logical operators. The logical operators are:-

.OR.	meaning logical disjunction, and
.AND.	logical conjunction.

The logical expression `a.AND.b` is `TRUE` if and only if both the logical expressions `a` and `b` are `TRUE`. The logical expression `a.OR.b` is `TRUE` if and only if at least one of the logical expressions `a` and `b` is `TRUE`. (i.e. it is an *inclusive or*). `.AND.` has precedence over `.OR.`. Arithmetic operators have precedence over relational operators, which have precedence over logical operators. If in doubt, use brackets. For example, the logical expression

```
jill .EQ. jack*2 .OR.  jack .GT. 0 .AND. jack .LE. 100
```

is equivalent to

```
(jill .EQ. (jack*2)) .OR. ((jack .GT. 0) .AND. (jack .LE. 100))
      2         1     6          3        5          4
```

The numbers show the order of evaluation.

4.3 Logical If Statement

A logical `IF` statement is of the form

IF (logical expression) S

where the logical expression is as described above, and S is any executable statement except a `DO` statement, another logical `IF` statement or any block-If command. The logical expression is evaluated, and if it has the value `TRUE`, the statement `S` is executed. If the value of the expression is `FALSE`, the statement `S` is not executed. N.B. In either case, control then passes to the next statement.

4.3.1 Examples of Logical If Statements

1. If **pay** is less than 4000.00, then increase **pay** by 275.25

```
IF (pay .LT. 4000.0) pay = pay + 275.25
```

2. Read a line of input data. If the integer on it is equal to 7, then read another line.

```
READ (5,*) number
IF (number .EQ. 7) READ (5,*) num2
```

3. Read a line. If the number on the line is 99 then stop.

```
READ (5,*) num
IF (num .EQ. 99) STOP
```

4. Test if **salary** is in the range 50,000 to 100,000 and write out a message saying 'Fat Cat' if it is.

```
      IF (salary .GE. 50000.0 .AND. salary .LE. 100000.0)
     $   WRITE (6,*) 'Fat Cat'
```

4.3.2 A Common Error With If Statements

How do you execute one of two statements depending on whether a test is true or false ? Suppose that you want to test to see if **pay** is less than **starve**. If it is, then add 250.0 to **pay**, but if it is not, then only add 100.0 to **pay**. One solution, which is incorrect, may be

```
IF (pay .LT. starve) pay = pay + 250.0
pay = pay + 100.0
```

This will not work because if **pay** was less than **starve** then we would add on 250.0 to **pay**, but then we would also add on 100.0. It is true that it would work in the case of **pay** being greater than (or equal to) **starve**, as we would only add 100.0 . So, another attempt at a solution leads us to another erroneous answer

```
IF (pay .LT. starve) pay = pay + 250.0
IF (pay .GE. starve) pay = pay + 100.0
```

Why is this wrong? It seems to work: if **pay** is less than **starve** then we will add on 250.0 and then we test again and only if **pay** is greater than or equal to **starve** do we add on 100.0 . This works sometimes, but if **pay** lies between **starve** and **starve - 250.0** then **pay** will be increased by 350.0! Try it in the case of **starve** having the value 4000.0 and **pay** having the value 3950.0 . The following solutions will work

```
IF (pay .LT. starve) addon = 250.0
IF (pay .GE. starve) addon = 100.0
pay = pay + addon
```

Compare this with

```
IF (pay .LT. starve) pay = pay + 150.0
pay = pay + 100.0
```

or

```
IF (pay .GE. starve) pay = pay + 100.0
IF (pay .LT. starve) pay = pay + 250.0
```

4.4 Block-If Statements

The logical IF statement above is limiting in that only one statement is allowed after the logical expression on the IF. 1977 standard Fortran introduced a construction known as a **block-IF**, that allows several Fortran statements to be executed as a result of one logical test. Further, it incorporates an ELSE mechanism which can be used to specify a block of statements that are to be performed if the logical test comes out false.

4.5 Block-If Terminology

Block-IF — The name of the IF group of statements. It starts with an IF.....THEN statement (see below) and ends with an END IF statement.

IF-block — The group of lines that lie between an IF.....THEN and the next ELSE IF, ELSE or END IF statement.

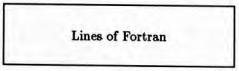
ELSE-IF-block — The lines that lie between an ELSE IF statement and the next ELSE IF, ELSE or END IF.

ELSE-block — The lines between an ELSE statement and the following END IF statement.

4.6 Basic Block-If

The form of the block-IF is

IF (logical expression) THEN



The IF-Block

END IF

The IF....THEN statement must appear on a line of its own. A statement cannot be put after the THEN on the same line! The END IF statement must also be put on a line of its own. The logical expression is evaluated. If it has the value of TRUE

then all the lines of Fortran in the If-block are executed. If the value of the logical expression is **FALSE** then the program skips the If-block and continues execution with the statement immediately following the **END IF** statement. Any executable statement or comment may appear within the If-block, including another block-If. In this case of nested block-Ifs there must be an **END IF** that corresponds to each **IF.....THEN**

4.6.1 Examples

1. Read in a value for **salary** and if it is negative then write out a message and stop the program. If it is non-negative, then write it out and go on with the program.

```
READ (5,*) salary
IF (salary .LT. 0) THEN
    WRITE (6,*) ' Negative salary is illegal ',
$               ' A value of ', salary,' was read'
    STOP
END IF
WRITE (6,*) ' Salary: ', salary
```

2. Read in a value for **num**. If **num** equals 0 then read in another value for **num**, write out a message saying that has been done and calculate values for **half** (half of **num**) and **twice** (2 times **num**).

```
READ (5,*) num
IF (num .EQ. 0) THEN
    READ (5,*) num
    half  = num/2.0
    twice = num*2
    WRITE (6,*) 'Second value of ', num, ' read'
END IF
```

   In this program segment, if **num** was originally read in as non-zero then **half** and **twice** would not have values.

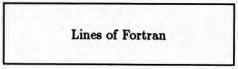
3. Same exercise as above, but if the second value of **num** is less than 10, then do not calculate **half** or **twice**.

```
READ (5,*) num
IF (num .EQ. 0) THEN
    READ (5,*) num
    IF (num. GE. 10) THEN
        half  = num/2.0
        twice = num*2
    END IF
    WRITE (6,*) 'Second value of ', num, ' read'
END IF
```

4.7 Else-If Block

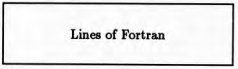
Any number of **ELSE IF** blocks may be in a block-If. The form is

```
IF (logical expression-1) THEN
```



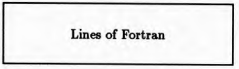
An IF-Block

```
ELSE IF (logical expression-2) THEN
```



An ELSE IF-Block

```
ELSE IF (logical expression-3) THEN
```



An ELSE IF-Block

```
   :
   . etc
END IF
```

There is only one **END IF** and that **END IF** corresponds to the whole block-If. The use of **ELSE IF** within a block-If is to test a logical expression and if it is true then execute the commands in the Else-If-Block. If the logical expression is false, then go to the next **ELSE IF** and test that expression. When eventually a value of true is found, the corresponding block of statements is executed, and then the program skips to the **END IF** statement. More exactly

1. Logical expression-1 is evaluated
2. If it is **TRUE** then
   (a) the statements in the If-block are executed.
   (b) the program skips all the **ELSE IF** blocks and continues execution following the **END IF**.
3. If logical expression-1 is **FALSE** then skip to the next **ELSE IF** (or **END IF** if there isn't an **ELSE IF**) and evaluate the logical expression there (logical expression-2).
4. if the logical expression is **TRUE** then

(a) the statements in the **ELSE IF** block are executed.

(b) the program skips to the **END IF**

5. If the logical expression is **FALSE** then skip to the next **ELSE IF** (or **END IF** if there isn't an **ELSE IF**) and evaluate the logical expression there.

6. Repeat steps 4 and 5 until eventually arriving at an **END IF**.

Any of these blocks can contain other (nested) block-Ifs, but one and only one **END IF** statement exists for each. That is, there is an **END IF** statement that corresponds to each **IF....THEN**, but not to any **ELSE IF....THEN** statements.

4.7.1 Example

Read a number. Take the following action depending on its value.

Value	*Action*
1	Read in 3 numbers and write out their average.
2	Read in 2 numbers and write out the result of raising the first to the power of the second.
4	Read in 2 numbers and if the second number is zero, then write out an error message and stop. If the second number > 0 then calculate the value of the first divided by the second.

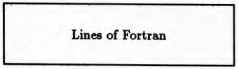
All input is integer.

```
READ (5,*) num
IF (num .EQ. 1) THEN
    READ (5,*) i1, i2, i3
    ave = (i1+i2+i3)/3.0
    WRITE (6,*) ' Average: ', ave
ELSE IF (num .EQ. 2) THEN
    READ (5,*) i1, ipower
    ival = i1**ipower
    WRITE (6,*) ' Power calculation: ', ival
ELSE IF (num .EQ. 4) THEN
    READ (5,*) n1, n2
    IF (n2 .GT. 0) THEN
        iquot = n1/n2
        WRITE (6,*) ' Integer quotient: ', iquot
    ELSE IF (n2 .EQ. 0) THEN
        WRITE (6,*) ' Divisor of zero found'
        STOP
    END IF
END IF
```

4.7.2 Else Block

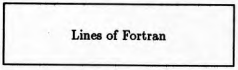
The form of an Else statement is the word **ELSE** on a line of its own. The Else block (if present) must come after all Else If-Blocks (if there are any). Execution of the Else block will be done if all logical expressions (at that level of nesting) evaluate to **FALSE**. The most complex form of a block-If is

```
IF (logical expression-1) THEN
```



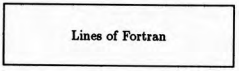
An IF-Block

```
ELSE IF (logical expression-2) THEN
```



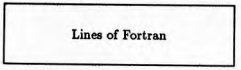
An ELSE IF-Block

```
ELSE IF (logical expression-3) THEN
```



An ELSE IF-Block

```
   :
   : etc
ELSE
```



An ELSE Block

```
END IF
```

The Else block will only be executed if logical expressions 1, 2, 3 etc all evaluate to **FALSE**. That is, each logical expression is tested and if one is found that is **TRUE**, then the corresponding block is evaluated. Otherwise, the Else block is evaluated.

A simple example using **ELSE** is

```
IF (pay .LT. starve) THEN
  pay = pay + 250.0
ELSE
  pay = pay + 100.0
END IF
```

This is the nicest solution to the problem discussed earlier in the chapter.

4.8 Notes on Block-If in General

1. To make programs easier to understand, use indentation within a block-If. This way statements that fall into the block stand out from those that are not in the block. Use an indentation of say 3 characters for each level of nesting.
2. Any executable statement, comment or **FORMAT** statement (to be done later) is allowed within any of the three different types of blocks. A statement must be wholly contained within a block.
3. In the next chapter, **DO** loops are covered. If a **DO** loop appears in a block, it must be wholly contained in that block.
4. Chapter 7 introduces a **GO TO** statement, which transfers control to another part of the program. You are not allowed to **GO TO** a statement within a block from outside that block.
5. Do not use a block-If statement if there is only one thing that is to be done as a result of a test. In such cases use a simple logical If statement.
6. Try to avoid very large block-Ifs. Large ranging block-Ifs make it difficult to keep track of where you are in a program.
7. Try to avoid very deep nesting of block-Ifs (you are allowed 25 nested levels).

4.9 Exercises

Exercise 4A

Identify the following relational expressions as valid or invalid.

1. `mpx .LE. 19+k`
2. `lax .GT. ama`
3. `cob .LT. .EQ. corn`
4. `14 .NE. ls`
5. `.EQ. 6`
6. `(I+19)*k .EQ. j*l*(jj+i)/k`
7. `p. GRT. q`
8. `l .=. 77`

Exercise 4B

Write a program to do the following — Read a line containing two integers. Determine which is the larger one and write out the two integers with the larger one appearing first.

Exercise 4C

Rewrite the following using block-IF statement(s).

```
READ (5,*) maxin
IF (maxin .EQ. 16) WRITE (6,*) 'Gotit'
IF (maxin .EQ. 19) forget = 0.5
IF (maxin .EQ. 10) forget = 0.0
IF (maxin .EQ. 16) STOP
IF (maxin .EQ. 19) WRITE (6,*) ' Found one'
```

Exercise 4D

Write a logical If statement that will test if A is in the range (-0.00001,0.00001) and if it is, set A to 0.0 .

Exercise 4E

Rewrite the following using only one level of block-If (and so only one `END IF` statement).

```
IF (ind .EQ. 16) THEN
    k = 6
    i = 9
ELSE
    IF (l .GT. j+4) THEN
        x = 19.6
        READ (5,*) y
        l = 0
    ELSE
        IF (main .LT. i) THEN
            cost = 19.0
            try  = 14.2
        ELSE
            pay  = 0.0
        END IF
    END IF
END IF
```

4.9.1 Exercise 4F

Write a program, complete with comments and informative input and output, to do some simple arithmetic for someone who runs the program. The program should ask the user to enter one of the following codes

1 to add two numbers
2 to subtract the second number from the first number

3 to multiply two numbers
4 to divide the first number by the second number

The program should then ask the user to enter the two numbers to be operated on, and then do the operation and print the answer.

Chapter 5

Do Loops

The Do-loop is a mechanism which enables repeated execution of a block of code (i.e. a group of Fortran statements) a number of times, without having to write the group of statements repeatedly. An example of a simple Do-loop is

```
      DO 25 i = 1, 6
      READ (5,*) num
25    WRITE (6,*) num
```

This loop begins with the Do statement and finishes with the statement numbered 25. The loop is executed 6 times, with the variable `i` taking the value 1 the first time around the loop, 2 the second time etc. and finally, 6 the last time. The statements in the loop are executed each time. That is, a number is read in and written out again, 6 times. 6 lines of data will be read. The variable `num` is re-used for each new number that is read in.

5.1 Do Statement

The complete syntax of the Do statement is

DO n i = m1,m2,m3

where

1. `n` is the statement label of the last statement of the Do-loop, which must be physically later in the program than the corresponding Do statement. The range of a Do-loop is the block of statements following the Do statement, up to and including the statement labelled `n`.

2. `i` is an integer or real variable and is called the Do-variable. The value of `i` may not be changed within the range of the Do-loop.

3. `m1`, called the initial parameter, `m2`, called the terminal parameter, and `m3`, called the increment parameter, are each an integer or real expression. `m3` is optional and it (and its preceding comma) may be omitted. In that case a value of 1 is assumed for `m3`. At the time of execution of the Do statement `m3` must not equal 0. `m1`, `m2` and `m3` may be changed during execution of the loop, but this will not change the number of times that the loop is executed or the increment used.

5.1.1 Evaluation of the Do Statement

A Do statement is used to define a loop. The action following the execution of a Do statement is described in the following steps.

1. The initial parameter, the terminal parameter and the increment parameter are converted to the type of the Do-variable, and the Do-variable is given the value of the initial parameter.

2. An internal iteration counter is established and is given the value of `(m2-m1)/m3 + 1` truncated to an integer. If this value is negative, the iteration counter is set to zero.

3. The iteration counter is tested. If the iteration counter is zero, then the Do-loop becomes inactive, and the program continues with the statement immediately following the last statement of the loop. If the iteration counter is greater than zero, then all the statements within the loop are executed.

4. At the end of this iteration of the loop, the value of the Do-variable is incremented by the increment parameter (`m3`), and the iteration counter is decremented by 1.

5. Go back to step 3, and so on around the loop until eventually the test in step 3 leads to the Do-loop becoming inactive.

So effectively, a Do loop will repeat all the statements from the one immediately after the Do statement up to and including the loop's last statement. This will be done a number of times, determined by `m1`, `m2` and `m3` (bearing in mind that the number of times could be zero).

5.1.2 Final Value of the Do-variable

When the range of a Do loop is exhausted (i.e. when the loop variable has a value that exceeds the terminal parameter on the Do statement), then control is passed 'out the bottom' of the Do loop. When this occurs, the value of the Do-variable retains its last defined value, which is the value after the last increment, when the iteration count is zero.

5.1.3 Restrictions on the Last Statement of a Do-loop

Some commands are not allowed to be the last statement of a Do-loop. These commands are

```
GO TO
DO
block-IF
ELSE IF
ELSE
END IF
RETURN
STOP
END
```

5.2 Continue Statement

The Continue statement is an executable Fortran statement which does nothing. When a Continue statement is executed, its effect is a 'no operation', and the program will simply go on to execute the next statement. The main use of the Continue statement is as the last statement of a Do-loop. It is a good idea to always end your Do-loops on a Continue statement. This way

1. The loop is easier to alter
2. The last statement stands out more
3. The last statement is legal.

5.3 Examples of Do-loops

1. Read 3 lines of data and write them out. This will produce 3 lines of output.

```
      DO 20 i = 1,3
         READ (5,*) val1,time,limit
         WRITE(6,*) ' Input values were:', val1,time,limit
20    CONTINUE
```

2. Write all the even numbers between 1 and 100 inclusive.

```
      DO 30 k = 2,100,2
         WRITE (6,*) k
30    CONTINUE
```

3. Read an integer from a line of data into the variable `int`. Now read `int` more lines and write them out. This is a very common method used to enable a program to process a variable amount of data.

```
      READ (5,*) int
      DO 40 i = 1,int
         READ (5,*) number
         WRITE (6,*) number
40    CONTINUE
```

4. Read up to 25 lines of data and write them out, until a line containing -9 is found, and then stop. This is another method of determining when there is no more data to read.

```
      DO 50 i = 1, 25
         READ (5,*) number
         IF (number .EQ. -9) STOP
         WRITE (6,*)' Input value is ', number
50    CONTINUE
```

5. Suppose that we read in some population statistics for the years from 1960 back to 1950 (in that order) and we wanted to write them out prefixed by the year to which they correspond.

```
      DO 60 iyear = 1960,1950,-1
         READ (5,*) ipop
         WRITE (6,*) ' In ',iyear,' the population was ',ipop
60    CONTINUE
```

This would write out

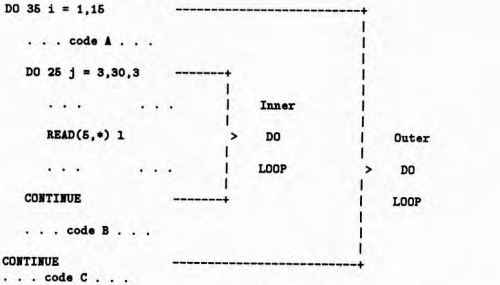
```
In    1960  the population was   XXXXXXXX
In    1959  the population was   XXXXXXXX
In    1958  the population was   XXXXXXXX
etc
```

6. Read 10 lines each containing an integer, and print each number. When a line contains the number -7, stop after printing the number.

```
      DO 80 i = 1,10
         READ (5,*) number
         WRITE (6,*) 'Input number = ', number
         IF (number .EQ. -7) STOP
80    CONTINUE
```

5.4 Nested Do Statements

It is possible to have a Do loop wholly contained within another Do loop. This is known as 'nesting Do loops.' For example,



The *inner* loop (down to statement number 25) is said to be nested within the *outer* loop (which ranges down to statement number 35). The operation of this example is as follows.

1. Set `i` to its initial value (1)
2. Calculate the iteration counter for the outer `DO`

3. Test the iteration counter. If > 0 then go on. If ≤ 0 then go to step 12.
4. Execute the code marked A
5. Set `j` to its initial value
6. Calculate the iteration counter for the inner `DO`
7. Test the iteration counter. If > 0 then go on. If ≤ 0 then go to step 10.
8. Execute the code in the inner `DO`
9. At label 25:
   (a) increment `j` by 3
   (b) decrement the iteration counter of the inner `DO`
   (c) go to step 7 above.
10. Execute the code marked B
11. At label 35:
   (a) increment `i` by 1
   (b) decrement the iteration counter of the outer `DO`
   (c) go to step 3 above.
12. Execute code marked C

How many lines will be read in this example ?

5.4.1 Rules for Nested Do-loops

An inner `DO` must terminate *on or before* the last statement of the outer `DO`. This means that Do-loops must not 'cross over'. The following nesting construction is *illegal.*

```
      DO 70 i = 3, 17
      . . . . .
      DO 80 j = 2,37,3
      . . . . .
70    CONTINUE
      . . . . .
80    CONTINUE
```

The following construction, where the inner and outer Do loops finish on the same statement, is legal.

```
      DO 90 j = 1,12
      DO 90 k = 3,17,2
      . . . . .
90    CONTINUE
```

In this example the inner Do will be completed before control is returned to the outer Do, even though they have the same last statement number. There is a limit to the depth of nesting of Do loops. This limit is not defined in the ANSI standard, but most computers allow at least 5 levels, and usually a lot more, although very few programs ever need more than a depth of 3.

5.4.2 Example 1

Suppose we want to find out the average salary earned in different electorates. We have a number of electorates to process and a variable number of people in each electorate. The program below will first read in a line of data indicating how many electorates there are. Following this there will be sets of salary data for each person in each electorate, one salary per line of data. The first line in each electorate set indicates the number of people in that electorate, and then following this line will be that number of salary lines. Each salary is a real number. For example if there were 3 electorates, with the following numbers of people in each:

electorate 1	2 people	earning	$256.50
			$291.85
electorate 2	4 people	earning	$196.45
			$202.44
			$180.00
			$175.10
electorate 3	3 people	earning	$120.45
			$133.22
			$110.90

Then the data would look like

```
3
2
256.50
291.85
4
196.45
202.44
180.00
175.10
3
120.45
133.22
110.90
```

and the program would be

```
C     Author:       G. Mander
C
C     Date:         November 1979
C
C     Purpose:
C
C      To find the average salary of people in a number of
C      electorates. The average in each electorate is printed,
C      and at the end the average of all salaries is printed.
C
C
C     Input Data:
C     -----------
C
C      All input is free format
```

```
C
C     (A)   First line
C           contains the total number of electorates (integer)
C
C     (B)   Following lines
C           contain for each electorate :
C           i) the number of people in the electorate (npeep),
C           ii) npeep salaries, one real number per line.

C       Variables Used:
C       ---------------
C         GRAND       the total salary earned by all
C                     the people
C         NELECT      the number of electorates
C         NPEEP       the number of people in an electorate
C         NPOP        the total number of people in all
C                     electorates
C         SALARY      the salary earned by a person
C         TOTAL       the total salary earned by an electorate
C
C
C

      READ (5,*) nelect
      grand = 0.0
      npop  = 0
      DO 20 i = 1, nelect
C
C        Read the number of people in each electorate

         READ (5,*) npeep
C
C        Read and total the salaries in an electorate
C
         total = 0.0
         DO 10 j = 1, npeep
            READ (5,*) salary
            total = total + salary
10       CONTINUE
C
C        Form the average for this electorate
C
         aver = total/npeep
         WRITE (6,*) 'The average for electorate ',i,' is ',aver
C
C        Add to grand total
C
         grand = grand + total
         npop  = npop  + npeep
20    CONTINUE
C
C     Calculate and write out grand average
```

```
C
      gave = grand/npop
      WRITE (6,*)
      WRITE (6,*) 'Average of all',nelect,' electorates is ',gave
      STOP
      END
```

The output from this program, using the data above would be

The average for electorate	1 is	274.17500
The average for electorate	2 is	188.49750
The average for electorate	3 is	121.52333
Average of all	3 electorates is	185.21222

5.4.3 Example 2

Write out a series of headings for a monthly diary. There is to be a one line heading for each month of each year between 1975 and 1982.

```
C
C      Author:  H. Mood
C        Date:  October 1979
C       Input:  There is no input
C     Purpose:
C        To write out a heading saying the year and month for
C        each month between 1975 and 1982
C
C
      DO 15 iyr = 1975,1982
         WRITE(6,*) '------------------------'
         DO 10 month = 1,12
            WRITE(6,*) iyr, month
            WRITE(6,*)
            WRITE(6,*) '++++++++++++++++++++++++'
10       CONTINUE
15    CONTINUE
      STOP
      END
```

5.5 Exercises

Exercise 5A

Identify the following statements as being true or false.

1. Statement labels must be assigned sequentially.
2. The largest statement number is 99999.
3. Every Fortran statement must be assigned a statement label.
4. Statement numbers may be variable quantities.

5. Statement numbers do not have to start in column 1.
6. It is valid to assign the same statement label to several statements in a program.
7. A **CONTINUE** statement must be the last statement of a Do loop.
8. A **CONTINUE** statement may be used outside a Do loop.
9. A Do loop must finish before another one may start.
10. A **CONTINUE** statement must be labelled.

Exercise 5B

Identify the following statements as correct or incorrect Do statements.

1. `DO 8 kan = j,k,1`
2. `DO n k6 = 5,n,2`
3. `DO 5 iy = 1,12`
4. `DO 692 3 = 1,k`
5. `DO 99 n1234 = 1,3,k`
6. `DO 9 k = 1,9.5`
7. `DO 88 kirsh = 1,k+4`
8. `DO 2 l = s,2`
9. `DO 7453 fred = 1,n4a2,2`
10. `DO 222 k = j,6`
11. `DO i = 1,9,2`

Exercise 5C

What is written out by the following groups of statements

1.
```
      vp = 98.6
      j  = 16
      IF (vp .le. 62.3) WRITE (6,*) j
      j = 14
      WRITE (6,*) j
```

2.
```
      DO 16 kx = 1,30,4
      IF (kx .GT. 16) WRITE (6,*) kx
      lx = kx-1
16    IF (kx .LE. 13) WRITE (6,*) lx
```

Exercise 5D

Assuming that there is one integer per data line, how many lines will be read in the following?

1.
```
      DO 16 i = 1,3
         DO 16 j = 1,4
16    READ (5,*) l
```

2.
```
      DO 16 i = 1,3
         DO 14 j = 1,4
            READ (5,*) l
14       CONTINUE
16    CONTINUE
```

3.
```
      DO 16 k = 1,3
         DO 14 j = k,4
14          READ (5,*) l
16    CONTINUE
```

4.
```
      DO 16 k = 1,3
         READ (5,*) l
         DO 14 j = 1,4
14       READ (5,*) l
16    CONTINUE
```

5.
```
      DO 16 kk = 3,6,1
      DO 16 lm = 8,11,4
16    READ (5,*) l
```

Exercise 5E

What number would be written out in the following?

```
      DO 17 j = 17,38,9
         . . . . .
17    CONTINUE
      WRITE (6,*) j
```

Exercise 5F

Using a Do statement, add up all the even integers between 98 and 224 inclusive and write out the total.

Exercise 5G

Write a program to do the following — Read a line of data containing an integer indicating the number of lines to follow. Read in the rest of the data, one number per line, and determine the largest and smallest number. For example, if the data were

```
4
12.2
16.4
-7.1
4.4
```

the output would be

```
The largest  number was     16.4
The smallest number was     -7.1
```

Exercise 5H

If a stone is dropped into a well and is heard to strike the water after an interval of t seconds, the depth d in metres to the water level is given by

$$d = \frac{s^2}{g} + st - \frac{s^2}{g}\sqrt{1 + \frac{2gt}{s}}$$

where s is the speed of sound in metres per second and g is the acceleration due to gravity in metres per second squared. Also note that

$$\sqrt{x} = x^{\frac{1}{2}}$$

Write a program to calculate d for t = 1, 2 and 3 seconds using g = 9.807 and s = 331.6 . (Notes: Constants should not be calculated more than once. Units of measurement are important in some problems; e.g. do not mix metres and feet.)

Chapter 6

Supplied Functions, Logical Variables, and Declarations

6.1 Supplied Functions

There is a library of special functions available to any Fortran program. Any of these system-supplied functions is invoked by using the function name, followed by its list of parameters, as part of a Fortran statement. The parameter list is enclosed in brackets. For example, the function **ABS** will find the absolute value of a number. **ABS** expects one parameter, the name of the variable or constant whose absolute value is required. To find the absolute value of the variable `bill` and to store that value in the variable **fred**, the statement is

```
fred = ABS(bill)
```

This could also be achieved by using the following statements

```
fred = bill
IF (bill .LT. 0.0) fred = -bill
```

The use of the **ABS** function is a little clearer and certainly more concise. You should try to use functions extensively for this reason. A function is said to *return* a value. In the above example, the function **ABS** returns a value which is the absolute value of its parameter.

6.2 Generic Functions

Normally a function expects each of its parameters to be a particular type (ie. real or integer), and the result returned is of a particular type. However, 1977 standard Fortran introduced the concept of generic functions. These are special supplied functions that may be used with different data types as parameters to the function. Earlier versions of Fortran only allowed specific data types for any specific function, so, for example, the function to find the larger of two integers (**MAX0**) had a different name to the function that found the larger of two reals (**AMAX1**). Now, the generic name **MAX** may be used in either case. The only restriction is that parameter data types cannot be mixed. Later on we will see how you can write your own functions, and in that case you can only write specific functions, that must *always* have the

specific data types you expect as their parameters. The list below has some of the more common generic functions. The abbreviations mean

I type integer
R type real
P the type of the parameters used

Function Name	*Result Type*	*Number of Parameters*	*Description*
`INT`	I	1	Truncated value of its parameter.
`REAL`	R	1	Value of its parameter converted to type real representation.
`ABS`	P	1	Absolute value of its parameter.
`MOD`	P	2	Remainder after dividing p1 (the first parameter) by p2 (the second parameter). ie. `Remainder = p1-int(p1/p2)*p2`
`MAX`	P	> 1	Largest of its parameters.
`MIN`	P	> 1	Smallest of its parameters.

For a more complete list of functions see your Fortran manual.

6.3 More Functions

There are some more complicated functions such as the trigonometric functions sine and cosine that are used in the same way as the functions described above. Some of the more common ones are

`SIN(real)`	returns the sine of real number expressed in radians
`COS(real)`	returns the cosine of a real number expressed in radians
`EXP(real)`	returns the exponential of a real number
`SQRT(real)`	returns the square root of a positive real number
`LOG(real)`	returns the natural logarithm of a real number

Using a function to return a value is called a *function reference*. Note that some functions return real values (e.g. the function `REAL`) while others return integer values (e.g. `INT`). The *number* and *type* of the parameters in a function reference are important and *must* be exactly what the function expects. In the case of generic functions, the parameters may be real or integer (but not a mixture). The parameters in a function reference may be arithmetic expressions of the same type that the function expects. For example, to find

$$\sqrt{(x1-x2)^2+(y1-y2)^2}$$

the statement may be

```
ans = SQRT( (x1 -x2) ** 2 + (y1 - y2) ** 2)
```

Function references may be used wherever arithmetic expressions are legal, and have precedence over arithmetic operators. For example, to find the sum of the sine and cosine of the variable `A`, the statement would be

```
ans = SIN(a) + COS(a)
```

Another example may be to find the square root of the larger of two real numbers. This may be done by

```
ans = SQRT (MAX (val1,val2))
```

So a parameter in a function reference may be another function reference, as long as it is a different function.

6.4 Logical Variables

So far the only variable types we have introduced are integer and real. Integer variables can store integer numbers and real variables can store real numbers. Now, we introduce *logical* variables, which can store the logical values true and false. Logical constants are written as

`.TRUE.`	meaning true
`.FALSE.`	meaning false

A logical variable may be assigned one of the constant values `.TRUE.` or `.FALSE.`, or may be assigned the result of a logical expression. It may also appear in `IF` statements as part, or all of the logical expression in parentheses. For example, if `nogood` is a logical variable we can use it in

```
IF (nogood) WRITE(6,*) 'Error found in data'
```

The way to read the above statement is to say: 'If `nogood` is true then write out the message'. Note that the value of `.TRUE.` does *not* appear in the expression.

6.4.1 Declaring Logical Variables

Before using logical variables they must be declared by appearing on a `LOGICAL` declaration statement. These *declaration* or *type* statements should appear at the top of a program before any executable statements. They may be in any order if there is more than one. The form of a logical type declaration statement is

`LOGICAL` <list of variables>

For example

```
LOGICAL poor, mid, high, male, female
```

The variables appearing on the list are called logical variables and must be treated as such throughout the program. Note that the first letter of the variable name has no special significance as the variable appears on a type statement.

6.4.2 Assignment

Logical variables may be assigned values that are true or false. For example

```
LOGICAL swap, endata
swap = .FALSE.
IF (num .LT. 0) endata = .TRUE.
```

In this last case, if **num** was not less than zero then **endata** would not have a value. A much better way, that would give **endata** a value of **.FALSE.** if **num** was not less than zero and **.TRUE.** otherwise is

```
endata = num .LT. 0
```

6.4.3 Using Logical Variables

Suppose a program reads in a salary and that salary is classified as

```
     0 - 5000        poor
  5001 - 12000       medium
 12001 - 20,000      high
20,001 - 100,000     fat cat
100,001 onwards      suspect error
```

In a program we can classify this by

```
LOGICAL poor, mid, high, fatcat, susp
READ (5,*) iwages
poor = iwages .LE. 5000
mid  = iwages .GT. 5000  .AND. iwages .LE. 12000
high = iwages .GT. 12000 .AND. iwages .LE. 20000
fatcat = iwages .GT. 20000 .AND. iwages .LE. 100000
susp  = iwages .GT. 100000
```

and then later in the program we can test these values and take different action depending on the wage classification as

```
IF (poor)          taxhit = taxhit - 5.0
IF (mid .OR. high) taxhit = taxhit + 0.75
IF (fatcat)        taxhit = taxhit + 7.5
```

This way we localise the classification ranges and then can refer to them using meaningful names. If in the future we want to change this program by altering the range classifications then we only have to make the change in one place, no matter how many times we refer to the ranges. Another example is in the processing of sex. Suppose that we read in a classification of 1 meaning **female** and 0 meaning **male**.

```
LOGICAL female, male
   .      .      .
   .      .      .
READ (5,*) isex
female = isex .EQ. 1
male   = isex .EQ. 0
   .      .      .
```

```
later in the program

IF (female) jobfem = jobfem + 1
IF (male)   jobmal = jobmal + 1
```

6.4.4 The .NOT. Operator

The **.NOT.** operator is used to turn logical values (**.TRUE.** and **.FALSE.**) into their opposites. For example

```
LOGICAL pos, neg
pos = num .GE. 0
neg = .NOT. pos
```

Now if **pos** is **.TRUE.** then **neg** will be **.FALSE.**, but if **pos** was **.FALSE.** then **neg** will be **.TRUE.** The **.NOT.** operator can also be used within an **IF** statement. For example

```
IF (.NOT. endata) READ (5,*) more
```

The **READ** will be done only if **endata** is **.FALSE.** Consider the setting of **male** and **female** above. Suppose that we wanted to set the logical variable **errsex** to **.TRUE.** for any code other than 0 or 1, and to set it to **.FALSE.** otherwise. Then

```
errsex = .NOT. (male .OR. female)
```

This is equivalent to

```
errsex = .NOT. male .AND. .NOT. female
```

.NOT. has precedence over **.AND.**

6.5 Type Declaration Statements for Integers and Reals

So far, the IJKLMN naming convention has been used for distinguishing between integer and real variables. This convention may be overridden so that, for example, **abc** may be the name of an integer variable and **IM** the name of a real variable. A type declaration for numeric variables consists of one of the declarations **INTEGER** or **REAL** followed by a list of variable names separated by commas, specifying those variables as being of type integer or real. Some examples are

```
INTEGER b
REAL j
INTEGER i,abc,root8
REAL matrix,number,x
```

Type declarations must precede all executable statements. The variable **i** could have been omitted from the **INTEGER** statment and **x** from the **REAL** statement without effect. These names are already identified as integer and real respectively by their first letters. On the other hand, there is no harm in such 'unnecessary' inclusions in type declarations. This may help to guard against failure to give the correct type to a variable whose name does not agree with the IJKLMN naming convention. There are various arguments for and against violating the default naming convention. If

you stick to the I to N default typing, then it is easier to follow the program, from the point of view of mixed type arithmetic, relating I/O lists to data, and checking parameter types in functions. If you declare variables as being a particular type, then you can use more meaningful variable names to describe the contents of variables.

6.6 Exercises

Exercise 6A

Use function references to return answers to be stored in either **ians** or **ans** depending on whether the function returns an integer or real result.

1. find the square root of $b \times b - 4.0 \times a \times c$
2. find the sine of 2.4
3. find the larger of **j** and **larg**
4. find the largest of **a** and **big**
5. find the absolute value of **ecc**
6. find the absolute values of **i**, **m**, and **a**
7. convert **kkk** to real
8. find the square root of **int**
9. find the largest of **i** and **x**

Exercise 6B

Write a program to calculate the factorial of a number that is read in. Factorial n is defined to be

$$n! = n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1$$

For example, $5! = 5 \times 4 \times 3 \times 2 \times 1$ and $0! = 1$. The factorial of a negative number is undefined.

Exercise 6C

Replace the line(s) of Fortran following by a single line that will produce the same answer in the indicated variable

1. **m**

```
m = ij/k
m = ij - m*k
```

2. **small**

```
IF (a .LE. b) small = a
IF (a .GT. b) small = b
```

3. `warm`

```
qwerk = ion
yipe  = kan
warm  = qwerk/yipe
```

Exercise 6D

Write a program to find the average of a set of numbers which are read from input. Do this by first reading the number of numbers that follow, call it n, then read n more numbers and find their average.

Exercise 6E

For an unknown number of numbers, read the number of values that follow, then those numbers and find their average and standard deviation. The formula for standard deviation is

$$sd = \sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n} - \overline{x}^2}$$

Chapter 7

The GO TO Statement

Fortran statements are executed in order of occurrence, starting with the first executable statement. Each statement is then executed in turn, and some action is performed. The `GO TO` statement has the form

```
GO TO n
```

where `n` is the statement label of an executable statement. Execution of the `GO TO` statement causes the statement identified by the statement label `n` to be executed next. The program then continues on from that point. Thus the `GO TO` statement may be used to transfer control to a statement other than the next one in sequence.

7.1 Examples of Go To Statements

Example 1 — Suppose we want to find the average salary of people who are defined as low salary earners. A low salary is defined as one below $5,000. The data that we have to process is entered as one salary per line, and we are to process data until an *end of file* is encountered. The way of representing the end of the file is dependent upon the machine used.

One method for solving the problem (the *Algorithm*) may be outlined as

1. Initialise a salary total and worker number total to zero
2. Repeat for as many salaries as there are
   (a) Read in a salary
   (b) If the salary is greater than or equal to 5000 ignore it.
   (c) If the salary is less than 5000 then
      i. add it to the salary total
      ii. add 1 to the number of workers total
3. When all salaries are processed calculate the average

A program to do the above is

```
C
C  Author:    L. Yamaha
C  Date:      Sept 1979
C
```

```
C  Language: VAX Fortran
C
C  Purpose:
C
C     To calculate the average salary of people who earn less
C     than $5000.
C
C  Input Description:
C
C      Each line contains an integer in free format indicating the
C      yearly salary of a worker (in dollars).
C
      ntot = 0
      npeep = 0
C
C     Read in and process the workers
C
10    READ (5,*,END=20) isal
      IF (isal .LT. 5000) THEN
          ntot  = ntot  + isal
          npeep = npeep + 1
      END IF
      GO TO 10
C
C     End of data, now work out the average
C     Check for zero in case there are no low wage earners
C
20    CONTINUE
      IF (npeep .EQ. 0) THEN
         WRITE (6,*) 'No low income earners found'
      ELSE
         ave = ntot/real(npeep)
         WRITE (6,*) 'The average salary of the ',npeep,
     $               ' low income earners found is ',ave
      END IF
      STOP
      END
```

Notice that there is only one `GO TO` in the above program. If it were not for the use of the block-if statement there would probably be two more `GO TO`s.

Example 2 — Read an integer from a line of data and write it out. Continue this process until a line with -99 is read. When this happens, write out a message and then stop.

```
C
C   Author:   Dee
C   Date:     Sept 1973
C   Purpose:  To read integers from input data and to
C             list them out
C
C   Input Description:
```

```
C
C             Free format input
C             one integer per line, last integer must
C             be -99
C
C   Restrictions:
C             Only the last data line may contain -99.
C             If this is not possible, then change
C             the value of istop
C
      istop = -99
10    READ (5,*) int
C
C     Test for the end of data
C
      IF (int .EQ. istop) THEN
          WRITE (6,*) 'End of job '
          STOP
      END IF
      WRITE (6,*) int
      GO TO 10
      END
```

7.2 Using Go To Statements With Do Loops

The last statement of a DO loop cannot be a GO TO statement or a logical IF statement that contains a GO TO statement. It is not legal to jump into the middle of a DO loop from outside the DO loop. For example, the following is *not* allowed.

```
      DO 40 i =1,ifreq
         . . . . .
         . . . . .
20    CONTINUE
         . . . . .
         . . . . .
40    CONTINUE
         . . . . .
         . . . . .
      GO TO 20
```

It is possible to leave a DO loop by using a GO TO statement. If this is done, the control variable of the DO is defined and is equal to the most recent value attained. For example,

```
      max = 4
      low = 2
      DO 20 i = 1,4
         low = low+1
         IF (max .EQ. low) GO TO 30
20    CONTINUE
30    WRITE (6,*) 'The value of i is ',i
```

will print the line

```
The value of i is   2
```

7.3 Using Go To Statements With block-if

It is illegal to jump into the middle of a block-if (any of the three types of blocks) from outside a block. A `GO TO` that transfers control within a block is allowed as is a transfer of control from within a block to outside (provided that you do not `GO TO` the middle of another block or a `DO`).

7.4 Reachability of Statements

Every executable statement must be 'reachable' along some logic path of the program. If some statement is not reachable, then an error should occur. Some computers will warn you but proceed and try to execute the program anyway, or they may not allow an attempt at execution until the problem has been rectified. It is always best to eliminate all diagnostics from your program, even if they are only warnings. For example, in the program

```
      isum = 0
      DO 20 i = 1,12
20    isum = isum + i
      GO TO 60
      WRITE (6,*) 'Just a heading '
60    STOP
      END
```

the `WRITE` statement can never be reached.

7.5 Some Common Errors With Go To Statements

1. Two statements having the same label.
2. The label referred to by the `GO TO` is missing.
3. Having an unlabelled statement after a `GO TO` statement — this statement will be unreachable.
4. Generation of an infinite loop by putting `GO TO` an earlier statement without any proper 'escape line' in between, to get out of the loop. For example,

```
(a) 10    i = i + 1
          i = i - 1
          GO TO 10
(b)       i = 5
    10    i = i+1
          IF (i .EQ. 0) GO TO 20
          GO TO 10
    20    CONTINUE
```

7.6 When and How to Use the Go To

A program is easiest to follow and thus easier to get running successfully if it has 'forward control' which refers to branches always going down the program. This is sometimes not possible, nor desirable, as shown in the first example in this chapter. Given that there are valid cases for using the `GO TO` there are still different program designs available, some of which are considered better than others. For example, consider the two blocks of code below (which are equivalent)

```
      IF (pay .GE. 6500.0) GO TO 10
      GO TO 20
10    pay = pay + 250.5
      nrich = nrich + 1
20    CONTINUE

      IF (pay .LT. 6500.0) GO TO 20
      pay = pay + 250.5
      nrich = nrich + 1
20    CONTINUE
```

By reversing the sense of the test in the logical `IF` the second block of code contains one less `GO TO` and one less statement label. Of course a much better way of doing this is to avoid using `GO TO`s entirely, with

```
      IF (pay .GE. 6500.0) THEN
          pay = pay + 250.5
          nrich = nrich + 1
      END IF
```

7.7 Comments and Go To's

The `GO TO` statement may transfer control a long way away from where the test that caused the branch of control occurred. If this happens, then it is desirable to put a comment above the statement gone to detailing why it is that you have come here. For example, suppose a program reads in data lines each containing an integer. If a value of 1 is read, then this will indicate that the data lines following are personnel records. If a value of 2 is read this will indicate that the lines following are inventory data.

```
C  Read an indicator variable
C
      READ (5,*) itype
C
C     Test the type
C
      IF (itype .EQ. 1) GO TO 25
      IF (itype .EQ. 2) GO TO 30
          .     .     .     .
          .     .     .     .
       test for further types
          .     .     .     .
          .     .     .     .
```

```
C
C     Type 1 record found, process personnel records
C
25    CONTINUE
        .   .   .   .
        .   .   .   .
C
C     Type 2 record found, process inventory records
C
30    CONTINUE
        .   .   .   .   .
```

7.8 Exercises

Some of these exercises do *not* require the use of `GO TO` statements. Where `GO TO` statements are not required, don't use them. It can become a bad habit to use `GO TO` statements excessively.

Exercise 7A

Write a program to evaluate

$$y = \begin{cases} \sqrt{a^2 + \sin^2 p} & \text{if } k < 0 \\ \sqrt{b^2 - \sin^2 p} & \text{if } k > 0 \\ \sqrt{a^2 + b^2} & \text{if } k = 0 \end{cases}$$

a, b, p and k are to be read from a line of data.

Exercise 7B

Write a program to evaluate the exponential of x for every integer from 1 to 100, printing x and its exponential side by side. Run the program and see what happens! (Note : exponential x is e^x — use the function `EXP`.)

Exercise 7C

For an unknown number of people, read an identification number and age (both integers) for each person. Print the maximum and minimum ages and the identification numbers corresponding to their first occurrences. Write an algorithm before writing the program.

Exercise 7D

Read the number of people in a survey. Then read an identification number and the age and sex of each person. Id, age and sex are integers. Male = 0, Female = 1. Print the id, age and sex (print `MALE` or `FEMALE`) of each person, and count the number of people over age 30. Write an algorithm before writing the program. Compartmentalize the problem, leaving difficult pieces of the problem to be solved properly later. Then write a more detailed algorithm.

Exercise 7E

Write a program to read x and evaluate

$$f = 1 + \cos x + \frac{\cos^2 x}{2!} + \frac{\cos^3 x}{3!} + \cdots + \frac{\cos^{20} x}{20!}$$

where $n!$ is called factorial n and is defined as

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

For example, $5! = 5 \times 4 \times 3 \times 2 \times 1$

Chapter 8

Subroutines

8.1 Subprograms

A program is a sequence of Fortran statements that is executed by a computer. The programs we have seen to date are referred to as *main* programs. Functions, such as the supplied functions, and subroutines, which we are about to study, are both called *sub-programs*. Main programs and sub-programs are called *program units*. When a main program is executed, the computer starts by executing the first line and then proceeds sequentially according to the rules we have learned. A subprogram is also a sequence of Fortran statements. Execution of a subprogram is initiated by the main program, when the main program refers to the name of the subprogram, which invokes or *calls* the subprogram. When the subprogram finishes, control goes back to the same or next statement in the main program and execution of the main program is continued. The main differences between subroutines and functions is the way they are invoked and where control is returned to. An example of a subprogram reference is the use of a standard generic function covered in Chapter 6. The statement

```
val = MAX(quant,zena)
```

is an example of a program calling the subprogram (in this case a function) named `MAX`. As well as referring to the subprogram by name, it may be required to supply a parameter list for the subprogram to use. This is done by enclosing the parameters in brackets after the subprogram name. The parameters provide a means of passing information to and from the subprogram. They are the only communication between the two program parts. Calling the function causes execution of the Fortran statements that comprise the body of the function. So far we have not been able to see the lines of Fortran involved, as the system keeps track of it all for the standard Fortran functions. Now we shall see ways in which to create our own subprograms for which the lines of Fortran that make up the body of the subprogram must be supplied by us.

8.2 Subroutines

Subroutines have names that are from 1 to 6 characters in length composed of alphabetic and/or numeric characters with the first one being alphabetic (these are the same rules as for variable names). Some examples of valid subroutine names are below.

```
ADDER
F47A62
J
READ
MOVE
```

8.2.1 Calling Subroutines

A subroutine is called by using the `CALL` statement, which has the form

```
CALL  name (parameter list)
```

where 'name' is the name of the subroutine. The parameter list is composed of constants, variables, function names and expressions, separated by commas. The subroutine is called with *actual* parameters. As in the case with the functions used so far, these actual parameters supply the subroutine with the actual values to be used. The subroutine may

1. reference a parameter (i.e. use its value),
2. change the value of a parameter, provided that the actual parameter is a variable name, or
3. give the parameter a value if it does not already have a value, provided that the actual parameter is a variable name.

When a subroutine is called, the actual parameters on the `CALL` statement are associated with the corresponding parameters in the subroutine. The parameters in the subroutine are called *dummy* parameters (or may be referred to as *formal* parameters). The list of dummy parameters is kept in the first line of the subroutine, called the subroutine header. When the subroutine is called, the dummy parameters become 'alias' names for the actual parameters. That is, they represent the same corresponding physical locations in the computer memory. The main program may call them one set of names (actual parameters), and the subprogram another set of names (dummy parameters). However, both refer to the same physical objects. The winner of the 1977 Melbourne Cup could be referred to as No. 2, or "Gold and Black", but it is the same horse. If there are no parameters then the brackets may be omitted.

8.2.2 Subroutine Statement

The form of the subroutine header is

```
SUBROUTINE  name (dummy parameter list)
```

The dummy parameters *must* be variable names which are associated with a list of actual parameters when the subroutine is called. The dummy parameter list may not contain constants or expressions. If the parameter list is empty, the brackets may be omitted. The association between actual and dummy parameters is in the order in which the parameters appear. The first actual parameter is associated with the first dummy parameter, the second actual parameter is associated with the second dummy parameter, etc. To make the association possible, both lists must correspond in

1. the number of parameters appearing in each list

2. the type of the parameters (integer or real)

The lines of Fortran within the subroutine (called the *body* of the subroutine) use the dummy parameters. When the subroutine is called, references within the subroutine to dummy parameters are really references to the corresponding actual parameters. The dummy parameters are so called because they do not exist as separate variables. No space in memory is reserved for them. They are just alias names for the physical locations of the actual parameters. For example,

```
tax = 1.6                SUBROUTINE sub(rose)
CALL sub(tax)                  .
      .                        .
      .                  xyz = rose/2.0
      .                  WRITE(6,*) xyz
      .                        .
      .                        .
```

would have exactly the same effect as

```
tax = 1.6
pqr = tax/2.0
WRITE(6,*) pqr
```

`rose` is just a name used by the subroutine to reference a real variable, which in this case happens to be known by another name in the main program. Since a dummy and actual parameter are the same location, any alteration in the value of a dummy parameter in a subprogram will also alter the value of the actual parameter in the main program.

8.2.3 A Subroutine for Getting Into a Car

Suppose that it is possible to write a subroutine in FORGLISH, which is a mixture of Fortran and English. The following may then be a subroutine for getting into a car.

```
SUBROUTINE  enter (car,door)
    1. open the DOOR of the CAR
    2. move through the DOOR into the CAR
    3. close the DOOR of the CAR
END
```

This Forglish subroutine would work equally well on any type of `car`. If it were called by

```
CALL enter (holden,door)
```

then the `car` referred to in the subroutine would actually be `holden`, as the actual parameter `holden` is associated with the dummy parameter `car`. In the call

```
CALL enter (porsche,door)
```

the `car` referred to in the subroutine would be `porsche`. In the call

```
CALL enter (datsun,boot)
```

the subroutine would now be used to get into the `boot` of `datsun`. In the call

```
CALL enter (f111,canopy)
```

the subroutine would now be used to get into an f111. What would happen in the call

```
CALL enter (gate,house)
```

Following through the subroutine, and associating the parameters together, we get gate being associated with car, and house being associated with door. and the subroutine is trying to

```
1. open the HOUSE of the GATE
2. move through the HOUSE into the GATE
3. close the HOUSE of the GATE
```

What has gone wrong? The associations are not what was intended. What was meant was house to be associated with car and gate to be associated with door. In order to do this we should have put

```
CALL enter (house,gate)
```

The order of the actual parameters is very important. The subroutine cannot tell if the order is 'correct' or not. It merely sets up the associations according to the order in which the parameters appear.

8.3 Return and End Statements

When a CALL is made to a subroutine the following events occur:

1. The association between actual and dummy parameters is made
2. The statements in the body of the subroutine are executed.

The subroutine must return control of execution back to the main program that called it, so that the main program may proceed with the lines of Fortran that follow the CALL. This is done by executing a RETURN statement in the subroutine, causing the execution of the main program to proceed from the line immediately following the CALL statement that caused us to get into the subroutine in the first place. There must be at least one executable RETURN statement in the subroutine. The need for conditional RETURN's may result in there being more than one RETURN statement in a subroutine. For example

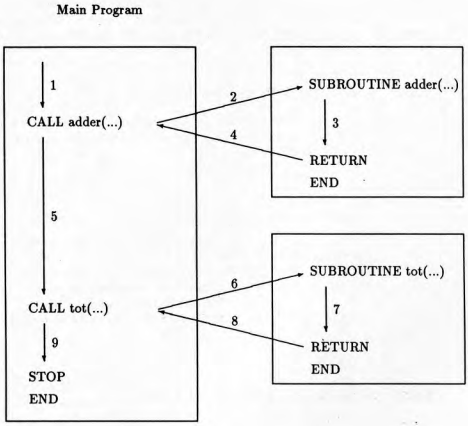
```
IF (name .EQ. myname) RETURN
READ (5,*) mesage
WRITE (6,*) 'Found mesage of ', mesage
RETURN
```

This part of the subroutine body says to return to the calling program in one of two ways. If the value of name is equal to the value of myname then return straight away. But if they are not equal, then read in a value for mesage and write it out before returning.

The last statement in a subroutine, or in fact in any program unit, must be an END statement.

8.4 Execution Paths Using Subroutines

The use of subroutines causes the execution path through the program to deviate at each **CALL**, to execute the body of the subroutine until it reaches a **RETURN** to the caller, which then proceeds on. This sequence of events may be shown by following the numbers on the arrows of the diagram below.



8.4.1 Program Structure

Many computers allow the subroutines to be set up so that they follow the main program in the one file. This structure is one way of organising the main program and subroutines and is not necessarily the way that it is done by all Fortrans, even on the same computer. It is usually allowable to have each program unit in a separate file. In any case, the main program and each subprogram has its own **END** statement.

8.5 Example of a Subroutine

The Post Office has given you directions to write a subroutine that will determine if a parcel may be posted, given the dimensions of the parcel and its weight. The criteria for acceptance are

weight < 10kg
length +2×(width+depth) < 100cm

We define the length as the longest side, depth as the smallest side, and width as the remaining side. It should be up to the program to sort out which side is which, so that all that the program user has to do is to input the three dimensions in any order, and also the weight.

8.5.1 Choosing Parameters

We need to have parameters to pass to the subroutine and also a parameter that will be returned from the subroutine, so that we will know if we can post the article or not. So we have the following

Name	*Type*	*Description*
SIDE1	Real	One of the dimensions of the parcel
SIDE2	Real	One of the dimensions of the parcel
SIDE3	Real	One of the dimensions of the parcel
WEIGHT	Real	The weight of the parcel in kg.
POST	Logical	Will be set by the subroutine to **.TRUE.** if the parcel can be posted **.FALSE.** if the parcel cannot be posted

8.5.2 Subroutine Header and Call

The subroutine header line will be

```
SUBROUTINE send (side1,side2,side3,weight,post)
LOGICAL post
```

Note that we must declare the type of **post** explicitly. The corresponding actual parameter in the main program will also have to be declared as a logical variable. The subroutine call will have 5 parameters. The first 3 will be real and represent the dimensions of the parcel, the next is real and represents the weight of the parcel, the last is logical and is an indicator that we can test to see if the parcel passed our tests. So, an example of the subroutine use will be

```
LOGICAL ok
READ (5,*) s1, s2, s3, weight
CALL send (s1,s2,s3,weight,ok)
IF (.NOT. ok) WRITE(6,*) s1,s2,s3,weight,' is not acceptable'
```

8.5.3 Subroutine Body

The subroutine has to first work out which dimension is which and then see if both criteria are satisfied.

```
      SUBROUTINE send (side1, side2, side3, weight, post)
      LOGICAL post
C
C     Work out which side is which. Length is the longest,
C     depth is the shortest, width is the middle one.
C
      al = MAX (side1, side2, side3)
      ad = MIN (side1, side2, side3)
      aw = (side1 + side2 + side3) - al - ad
      dim = al + 2.0*(aw + ad)
      post = (weight .LT. 10.0) .AND. (dim .LT. 100.0)
      RETURN
      END
```

Alternatively,

```
      al  = MAX (side1, side2, side3)
      dw  = (side1 + side2 + side3) - al
      dim = al + 2*dw
      ... etc.
```

One disadvantage of the above subroutine is that we don't know exactly what is wrong with the rejected parcel. We could overcome this in a number of ways

1. Write out a message in the subroutine to say what (if anything) is wrong with the parcel, and still return the value of `post` as above, so that we would know how to process that parcel further (if required).
2. Return one of a number of values (rather than just true or false) depending on what was wrong with the parcel. Then we would have to test this value in the main program and write out an appropriate message or take appropriate action.

As an exercise, re-write the subroutine to improve its output (i.e. make it more useful).

8.6 Using Subroutines

Subroutines may be referenced from other subroutines as well as from the main program. Care must be taken to ensure that a subroutine does not call itself, either directly or indirectly. An example of calling subroutine A indirectly would be

Main calls A, A calls B, B calls C, then C calls A again.

The significant characteristic of a subroutine is that it is quite independent. It can be written in isolation from the main program. The main program need know nothing about how it works. It needs to know only its specification (i.e. what it does and how to call it). This means that it is possible to set up *libraries* of subroutines, so that once a subroutine has been written for, say, solving linear simultaneous equations or calculating standard deviations or correlation coefficients, it may be made available to other interested programmers, to be used by them as larger building blocks, or simply to save them the trouble of writing their own versions.

8.7 Example 2 - Calculate the Area of a Triangle

Problem — To calculate the area of a triangle given the lengths of its sides, and also to determine if the triangle is reasonable. A reasonable triangle is one where the sum of any two sides is greater than the third. The steps involved in writing a subroutine to do this are

1. Choose a name for the subroutine, say, `tri`, and decide on the dummy parameters. Here the dummy parameters will be

Name	*Type*	*Description*
A	Real	One of the lengths of a side
B	Real	One of the lengths of a side
C	Real	One of the lengths of a side
AREA	Real	The area of the triangle
OK	Logical	Will be set by the subroutine to `.TRUE.` if the triangle is reasonable `.FALSE.` otherwise

So we arrive at the heading line

```
      SUBROUTINE tri ( a,b,c,area,ok )
```

2. Next we write the body of the subroutine, to perform the desired operations on the dummy parameters. The entire subprogram is then

```
      SUBROUTINE tri ( a,b,c,area,ok )
C
C     Author:    L. Landau
C     Date:      Sept 1977
C                Modified by L. Landau oct 1979
C
C     Parameter Description
C     ---------------------
C
C      Parameters with an * next to them are altered by this
C      subroutine
C
C        A       one side of the triangle              (real)
C        B       one side of the triangle              (real)
C        C       one side of the triangle              (real)
C  *     AREA    the calculated area of the triangle   (real)
C  *     OK      .TRUE. if the triangle is reasonable  (logical)
C                .FALSE. if the triangle is not reasonable
C
C     If the triangle is not reasonable an area of 0.0 will
C     be returned
C
C     Purpose:
C
```

```
C       To calculate the area of a triangle given its sides
C       and to test to see if the triangle is reasonable by
C       ensuring that the sum of any two sides exceeds the third
C
      LOGICAL ok
      IF ( a+b .LE. c) GO TO 5
      IF ( a+c .LE. b) GO TO 5
      IF ( b+c .LE. a) GO TO 5
C
C     Triangle is reasonable
C
      ok = .TRUE.
      s  = 0.5 * (a + b + c)
      area = SQRT( s*(s-a)*(s-b)*(s-c) )
      RETURN
C
C     Triangle is unreasonable
C
5     CONTINUE
      ok = .FALSE.
      area = 0.0
      RETURN
      END
```

3. This subroutine can now be used by any program to calculate the area of any given triangle specified by the program. So when writing such a program, the programmer need not worry about finding areas of triangles, but can simply write

   ```
   CALL tri( side1,side2,base,area,legal )
   ```

   where `side1, side2, base` have been given values in the main program at some point prior to the call, and `legal` has been declared as type logical.

8.8 Common Errors

It is important to note that there must be a one to one correspondence between actual parameters and dummy parameters. They must agree in

1. **Number of parameters** — There must be the same number of actual parameters as there are dummy parameters, for ANY call of the subroutine.

2. **Type of parameters** — There may be a mixture of real and integer parameters, but the respective types of the actual parameters must agree with the respective types of the dummy parameters. For example, if the subroutine has a first line of

   ```
   SUBROUTINE zot(i,adder,last,aver,many,zz,corr)
   ```

   then the actual parameters would have to be of type integer, real, integer, real, integer, real, real, respectively.

8.9 Exercises

Exercise 8A

Which of the following are legal statements ?

1. `CALL fred (why, 2, not, a+b*c)`
2. `CALL SUBROUTINE switch (x,y)`
3. `CALL environment (tree, leaf, 3*rock)`
4. `SUBROUTINE envir (tree, leaf, 3*rock)`
5. `SUBROUTINE action`
6. `SUBROUTINE mult (in1, in2, out)`
7. `IF (n .eq. 0) END`

Exercise 8B

Do any of the following statements correspond to a statement in Exercise 8A ?

1. `SUBROUTINE fred (why, num, i2, ans)`
2. `CALL action (speed)`
3. `CALL action`
4. `CALL mult (3*int, num*2, answer)`

Exercise 8C

Write a subroutine to return the value of y, given the value of x, such that

$$y = \frac{1}{x}$$

If x is zero, return zero for the value of y. Write a main program to call the subroutine with the following values of x : 5, 0, 0.5 .

Exercise 8D

Write a subroutine which, given an angle in degrees, e.g. 37.278, calculates degrees, minutes, and seconds as integers. Take the seconds to the nearest integer. There are 60 seconds in a minute and 60 minutes in a degree.

Exercise 8E

Write a subroutine to find the factorial of a number which is supplied to the subroutine. Return the answer (do not print it in the subroutine).

Exercise 8F

Write a program to calculate ${}^nC_r$ for integers n and r. The formula is

$$ {}^nC_r = \frac{n!}{r!(n-r)!} $$

where r is less than or equal to n, and n and r are both greater than or equal to zero. Zero factorial is 1. Use a subroutine to calculate the factorial of a number. The main program should call the subroutine three times.

Chapter 9

Fixed Format Write

So far, free field format has been used with both **READ** and **WRITE** statements. It is possible to specify, in a **FORMAT** statement, the layout of each line to be printed. In order to do this the asterisk in the **WRITE(6,*)** is replaced by a number which is the statement number of a **FORMAT** statement. The form of a **FORMAT** statement is

```
n     FORMAT  (field descriptors)
```

where 'n' is a statement label and the field descriptors specify how an output line is to be laid out into fixed length fields into which values are to be written. When using this type of output, headings and strings may no longer appear in the output list on the **WRITE** statement, but must be placed among the field descriptors in the **FORMAT**. The **FORMAT** statement is referenced when the corresponding **WRITE** statement is executed. A **FORMAT** statement may be referenced by several **WRITE** statements. The **FORMAT** statement itself is *non-executable*. It may appear anywhere within the program, but it is not good practice to terminate a do-loop with a **FORMAT**. It may appear either before, after or nowhere near its corresponding **WRITE** statement(s). The correspondence between the two statements is achieved via the statement number. For example,

```
      WRITE (6,103) k, a
103   FORMAT (field descriptors)
```

The **WRITE** statement lists the names of the variables to be printed and the **FORMAT** statement contains field descriptors which will describe the fields for an integer and a real number, as well as any headings. Field descriptors are inserted between the brackets of the **FORMAT** statement and are separated from other field descriptors by commas. A field descriptor defines the number of columns to be allocated to a number or character string, so the field descriptors consecutively allocate the columns across the line.

9.1 Field Descriptors

9.1.1 Blanks in the Output Line

Print blanks in the output line with the **X** field descriptor, which has the form

```
nX
```

where 'n' is the number of spaces to be placed in the printed line. The first character on an output line is not printed, but treated specially. For the moment, let us put `1X` as the first field descriptor so that the first character is a blank (which is not printed).

9.1.2 Writing Headings Using Fixed Format

The headings should appear within the **FORMAT** statement, enclosed in quotes. For example,

```
      WRITE (6,104)
104   FORMAT (1X, 'SHOP INVENTORY', 25X, 'GROCERY SECTION')
```

writes the line

```
SHOP INVENTORY                          GROCERY SECTION
```

Note that there is no output list on the **WRITE** statement, as no variables are to be written with the heading. Also note the difference between blanks in a Fortran statement and blanks to be printed in the output. Fortran statements may be freely spaced out to improve the readability of the program; whereas blanks are printed in the program output by using the `X` descriptor or by putting blanks in quotes in the **FORMAT** statement. So the above **FORMAT** is equivalent to

```
104   FORMAT (' SHOP INVENTORY',
     $        '                          GROCERY SECTION')
```

9.1.3 Integer Field Descriptor

The field descriptor for printing integers has the form

```
Iw
```

where

`I` is the letter `I` and indicates that an integer value will be printed in this field.

`w` is a number (that you must supply) and refers to the width of the field. The width of the field is the number of columns (or printing positions) allocated to the integer. It is the maximum number of digits that the number may have.

For example, the statements

```
      int = -27
      WRITE (6,100) int
100   FORMAT (1X,I5)
```

will print the line

␣␣-27

where ␣ indicates the blank character. The `1X` in the **FORMAT** statement is not printed as a space. The 5 characters (viz, ␣␣-27) come from the `I5` in the **FORMAT** statement. The number is printed right justified in the field with leading blanks to make up the required field width. If more than one value is to be written out, then there will be more than one variable in the output list, and there must also be a field descriptor to correspond to each variable in the output list. For example, if `men` has value 12345 and `lib` has value 0, then

```
      WRITE (6,100) men, lib
100   FORMAT (1X,I8,I4)
```

will produce the output

```
␣␣␣12345␣␣␣0
```

The `I8` is used to write out a value for `men` in the first 8 columns and the `I4` is used to write out a value for `lib` in the next 4 columns.

9.1.4 Real Field Descriptor

The field descriptor for real numbers has the form

```
Fw.d
```

where

F is the letter **F** and indicates that a real value will be printed in this field

w is a number and refers to the width of the field

d is a number indicating the number of decimal places to be printed.

For example, the statements

```
      vice = -33.47
      WRITE (6,110) vice
110   FORMAT (1X,F10.3)
```

will print the line

```
␣␣␣-33.470
```

Again, the `1X` is not printed. The ten characters (viz, `␣␣␣-33.470`) come from the 10 in `F10.3`. The three characters 470 come from the 3 in `F10.3`. Note that leading places are printed as blanks, trailing places are zeros, and columns for the minus sign (if there is one) and the decimal point are included in the field width. Formatting may be used to round off real numbers. This only affects the way they are printed, not the values stored in memory.

9.1.5 Mixing Real and Integer Field Descriptors

Of course it is possible to print out more than one number per line, and to have both real and integer numbers on the same line. When a mixture of reals and integers are written out in the one `WRITE` statement there must be an exact correspondence between the types of variables on the output list (integers or reals) and the types of field descriptors used (`I` or `F`). An `F` field descriptor in the `FORMAT` statement must correspond to a real variable in the output list in the `WRITE` statement, and an `I` field descriptor in the `FORMAT` statement must correspond to an integer variable in the output list in the `WRITE` statement. Field descriptors are separated by commas in the `FORMAT` statement, and variable names are separated by commas in the output list on the `WRITE` statement. For example, the statements

```
      ave   = 10.3
      small = -7.53
      num   = 23
      mint  = -10
      WRITE (6,120) ave, small, num, mint
120   FORMAT (1X, F7.2, F7.1, I5, I6)
```

will print the line

```
␣␣10.30␣␣␣-7.5␣␣␣23␣␣␣-10
```

and

```
      WRITE (6,140) ave, small, num, mint
140   FORMAT (1X, F7.2, 3X, F7.1, 1X, I5, 20X, I6)
```

will print the line

```
␣␣10.30␣␣␣␣␣␣-7.5␣␣␣␣23␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-10
```

9.1.6 Repetition of a Field Descriptor

Where two or more consecutive field descriptors are identical in every respect, then a shorthand notation may be used. This is accomplished by writing the field descriptor only once and prefixing it with a number indicating the desired number of repetitions. For example, the following **FORMAT** statements are equivalent

1. ```
   160   FORMAT (1X,I5,I5)
   ```
   and
   ```
   160   FORMAT(1X,2I5)
   ```
2. ```
   130   FORMAT(1X,F10.3,F10.3,F10.3,I4,I3,I3,F10.3)
   ```
   and
   ```
   130   FORMAT(1X,3F10.3,I4,2I3,F10.3)
   ```

9.1.7 Combining Headings with Numerical Output

The heading, or character string, is placed between quotes (as before) in the **FORMAT** statement, and will be printed in the next n columns, where n is the number of characters in the string. That is, extra spaces are not automatically put around the string. For example, the statements

```
      WRITE (6,150)
150   FORMAT (1X,'This is a heading')
      WRITE (6,160) ave, small, num, mint
160   FORMAT (1X,'Value ',F7.2,' Hi there ',
     *        F7.1, I5, I6, 5X, 'Message')
```

will print the lines

```
This is a heading
Value    10.30 Hi there    -7.5   23   -10     Message
```

Note that **5X** in a **FORMAT** statement is equivalent to '␣␣␣␣␣'.

9.2 Print Control Character

So far, the first field descriptor in a **FORMAT** statement has been **1X**. This first character of every output line has the special function of controlling paper spacing and is never actually printed. In standard Fortran, you can print 132 columns as the print control character is not part of the output image. The character that is in the first column of each output line is known as a print control character and has the following effect on the output

Character	*Vertical Spacing Before Printing*
␣ (blank)	one line (i.e. single spacing)
0 (zero)	two lines (i.e. double spacing)
1 (one)	skip to first line of next page (printer only)
+	no advance (i.e. print on the same line)

These characters are placed into the output line by putting them in quotes, i.e. **'␣'** (or **1X**), **'0'**, **'1'** and **'+'**. If a character other than one of those defined above is placed in column 1 of the output line, the result is unpredictable. A **'+'** should only be used to underline headings, and not to set up a line of numbers with different **WRITE** statements. When printing output at a terminal, you must still allow for the print control character.

Example — Print the values for variables **min** and **max** at the top of a new page. Then write a blank line followed by values for the variables **dollas** and **kost** on one line. Suppose the values have been set up as

```
min    = -126
max    = 2
dollas = 77.2576
kost   = 10301
```

then the following lines of program

```
      WRITE (6,170) min, max
170   FORMAT ('1', I6, 3X, I5)
      WRITE (6,180) dollas, kost
180   FORMAT ('0', F10.2, I7)
```

will print (on a new page)

```
␣␣-126␣␣␣␣␣␣␣2
␣␣␣␣␣␣77.26␣␣10301
```

9.2.1 Multi-line Format

A slash (i.e. /) in a **FORMAT** statement indicates that the current output line is complete and the next one is to begin. It does not need to be separated from field descriptors with commas. For example,

```
      WRITE (6,190) ave, small, num, mint
190   FORMAT (1X,2F10.3/1X,2I4)
```

will print the lines

```
    10.300     -7.530
 23 -10
```

Note that the **1X** after the slash is required as it will be at the start of a new line and is therefore taken as a print control character. If n consecutive slashes are written in sequence at the start of a **FORMAT** statement, n lines are left blank before printing the first line. If n consecutive slashes are written in sequence at the end of a **FORMAT** statement, n lines are left blank after printing the last line. If n consecutive slashes are written in sequence between other field descriptors in a **FORMAT** statement, n-1 lines are left blank between the two sets of printed lines.

9.3 Example of Formatted Write Statement

Suppose the desired output is

```
(new page)
GROSS INCOME = xxxxx.xx
TAX PAID     = xxxxx.xx
SUPER  = xxx.xx     DEDUCTIONS = xxx.xx
MONTHS OF SERVICE = xxx
```

where **x** represents a digit. This will require 5 variables, the first four will be real and the last an integer. Assume the variables have these values.

```
      gross = 6847.2
      tax   = 948.353
      super = 66.1
      ded   = 2.65
      month = 95
```

Then the **WRITE** and **FORMAT** statements are

```
      WRITE (6, 200) gross, tax, super, ded, month
200   FORMAT ('1GROSS INCOME = ', F8.2/
     $       1X, 'TAX PAID', 5X, '= ', F8.2/
     $       1X, 'SUPER  = ', F6.2, 5X, 'DEDUCTIONS = ', F6.2/
     $       1X, 'MONTHS OF SERVICE = ', I3)
```

It is not necessary to continue on to a new line of the **FORMAT** statement at the same point that a new output line is specified. it is not the continuation, but the slash, that causes a new output line to be taken.

9.4 Design of Format Statements

Format statements are often the source of errors as they can become quite complex. There are two things that may be done to simplify the task of format design.

1. Before starting to write out the **FORMAT** statement you should first draw up a skeleton of the output that you require, marking in all the headings and blanks that are required, as in the example above.

2. Use sensible continuation points. There is no need to go right up to column 72 before going to a new line. Pick an easy break between field descriptors at least.

3. Never continue a **FORMAT** statement (onto a continuation line) in the middle of a heading. Terminate the heading in a convenient place (e.g. on a word boundary) and then start again on the continuation line. So instead of

```
      WRITE(6,100)
100   FORMAT (1X, 'The number of politicians in the upper
     +house is limited')
```

put

```
      WRITE(6,100)
100   FORMAT (1X, 'The number of politicians in the upper ',
     +        'house is limited')
```

Note the space after the word **upper** and the comma between the two headings. The output in both cases would all be on the same line. In the first case, the first line of the **FORMAT** statement uses all 72 characters even though 'upper' ends in column 55, so 17 spaces will be printed between 'upper' and 'house'. Remember that any characters after column 72 are ignored.

9.5 Common Errors and Points to Note

1. Trying to print a number that is too large for the space allowed. Many computers will fill the entire field with asterisks. The statements

```
      oops = -6.235
      WRITE (6,210) oops
210   FORMAT(1X,F5.3)
```

will print the line

```
*****
```

2. Omitting the print control character. Remember that the first character of every output line is used for print control and is not printed. For example,

```
      i = 1234
      WRITE (6,220) i
220   FORMAT(I4)
```

will print the following line at the top of the next page

```
234
```

3. Using variables in a **FORMAT** statement.

Illegal : **FJ.K NI10**
Legal : **F5.3 3I10**

4. Trying to write an integer using an **F** field descriptor or a real using an **I** field descriptor. The order and type of variables in the list of the **WRITE** statement must correspond with the order and type of field descriptors in the **FORMAT** statement.

5. Trying to print more than 132 characters on a line. If this is attempted, an error occurs.

9.6 Exercises

Exercise 9A

Find the errors in the following Fortran statements and suggest ways in which they might be corrected.

```
1.        WRITE (6,230) k
    230   FORMAT (' Finished the job')

2.        WRITE (6,240) rolled,stones,gather,no,moss
    240   FORMAT (1X,2F10.2,2I5,F3.5)

3. 250    i = 17
          DO 250 k=3,49
          READ (5,*) i,f,k
          WRITE (6,270) i
    270   FORMAT (1X,F7)
          IF (i .=. 26) STOP
    280   CONTINUE
```

Exercise 9B

How many lines will be printed?

```
      WRITE (6,290) zap,bam,zowie,pow
290   FORMAT (1X,'A = ',/,' ',2F10.3,' B = ',F6.1/
     +       '0C = ',F9.1)
```

Exercise 9C

What is the output from the following?

```
      jill = 0
      zero = 0.0
      yours = 16.77
      tax = -586.21
      marvin = 90062
      min = -587
      jack = 10
      in = 123456
      out = 5.4827
      whynot = 131.1
      WRITE (6,300) jill,jack,min,marvin,in
300   FORMAT(1X, 5I5)
      WRITE (6,310) zero,yours,tax,out,whynot
310   FORMAT(1X, 5F7.2)
```

Exercise 9D

Indicate whether the field descriptors on the left are sufficient to print the numbers on the right and show what would be printed.

Field Descriptor	*Number*	*Printed Result*
F5.1	676.71	
I4	6381	
F10.1	132.63	
F4.2	12.16	
I10	99999	

Exercise 9E

Consider two blocks a and b. Block a is a cube of side length h and block b has sides k, $2k$, $3k$. Read in the values of h and k. Print your name and, after two blank lines, write the heading

```
Comparison Of Surface Areas
```

in the middle of the line. After three blank lines, write the following lines with their calculated values.

```
Side of cube A:
Width of block B:
Height of block B:
Length of block B:
Surface area of A:
Surface area of B:
Difference in surface area:
Larger surface area:
```

Exercise 9F

Write a program to evaluate the sine, cosine, tangent, secant (1/cos), cosecant (1/sin), and cotangent (1/tan) of every integral angle from 1 to 89 degrees inclusive. Print the output in tabular form, with headings.

Chapter 10

Fixed Format Read

Fixed field format may also be used with **READ** statements. The data line is divided into fixed length fields as defined by field descriptors in the **FORMAT** statement. The field descriptors will specify the positions on a data line that will contain the values to be read. This now means that values will no longer be separated by commas in the data, but must be exactly within the columns specified. As blanks take up columns, they are significant, and in numeric data they are interpreted as zeros. **FORMAT**s corresponding to **READ** statements do not require a print control character. Data is read from column 1 onwards.

10.1 Field Descriptors

10.1.1 Integer Field Descriptor

The integer field descriptor has the form

```
Iw
```

where `w` is an integer constant indicating the width of the field, which includes any + or − sign that may be present. Consider the statements

```
      READ (5,100) number, next
100   FORMAT (I5, I4)
```

If the input data line is

␣␣345␣-12

then **number** will be set to 345 and **next** will be set to −12. For integer field descriptors, blanks in the data are interpreted as zeros, so if the input is

␣345␣␣-12

then **number** and **next** will be set to 3450 and −12 respectively. To read in 4 integers that are laid out on one line as

Number	*Columns*	*Width*
1	1 − 4	4
2	5 − 10	6
3	11 − 17	7
4	18 − 19	2

the Fortran statements would be

```
      READ (5,122) jane, kinda, likes, nudes
122   FORMAT (I4, I6, I7, I2)
```

If the data line is

9876␣␣␣-129␣␣␣␣␣␣␣320

then the effect of the **READ** statement is the same as the four assignment statements

```
jane  = 9876
kinda = -129
likes = 3
nudes = 20
```

Warning : *Blanks in input data are interpreted as zeros.* The following statements

```
      READ (5,134) kool
134   FORMAT (I5)
```

read an integer right-justified in columns 1–5. If the data is

␣2

then **kool** will become the value 2000 .

10.1.2 Real Field Descriptor

The real field descriptor has the form

Fw.d

where

w — is an integer constant indicating the total width of the number (i.e. the total number of columns it takes up), including any decimal point or sign.

d — is an integer constant indicating how many columns are to be interpreted as following an implied decimal point. This 'd' is ignored if there is a decimal point in the number being read.

Consider the following statement

```
      READ (5,110) val, ewe
110   FORMAT (F10.3, F7.2)
```

and the data line

␣␣␣1234567␣␣-45892633

The first value (to be read into variable **val**) starts in column 1, and is 10 columns wide, the last 3 columns are to be interpreted as being *after* the decimal point. So, **val** will be given the value 1234.567 . The reading then continues from the next column. The next 7 columns will contain the value for variable **ewe**, the last 2 columns to be interpreted as that part of the number after the decimal point. So, **ewe** will have the value of –45.89 . Notice that the remaining columns are ignored, as there are no further variables on the input list. Remember, blanks are taken as zeros, and a decimal point appearing in the data will override the 'd' specification, in the **Fw.d** . For example, consider

```
      READ (5,123) vice, squad
123   FORMAT (F10.3, F7.2)
```

and the data line

␣␣47047.9␣␣-12␣␣␣2345

This will set **vice** to 47047.9 and **squad** to -120.0 .

10.1.3 Reading Integers and Reals

Integer and real values can both be read with the one **READ** statement. The **FORMAT** statement must reflect the types of variables that you are reading. It is an error to read in a real number using I format, or to read in an integer using F format. These will either cause the program to terminate or to give incorrect results. Another common error is to forget to ensure that there is a correct correspondence between the number and type of format descriptors used, and the variable names in the input/output list. If this is not the case, then the computer may not tell you of an error, but will not store the number you want in the variable on the input list, or will not write out the correct value of a variable on the output list. Just remember that reals and integers are stored differently in memory. For example, suppose we wish to read 5 numbers from a line of data; the first 2 real, the next 2 integer, and the last one real. Now we will have to reflect this in *both* the **READ** input list, and in the **FORMAT**. So let us first choose variable names of

```
great, blue, movies, never, fail
```

Notice that the first two are real, the next two are integer, and the last is real. Now before we can go any further, we must specify where on the line of data the numbers are to go.

Variable	*Columns*	*Width*
great	1 – 7	7
blue	8 – 16	9
movies	17 – 24	8
never	25 – 30	6
fail	31 – 40	10

Assume that if there is no decimal point in a real number that we want the number to be interpreted as having 1 digit after the decimal point. Then, consider the statements

```
      READ (5,211) great, blue, movies, never, fail
211   FORMAT (F7.1, F9.1, I8, I6, F10.1)
```

and the following line of data

␣␣23.44␣-8.23899␣␣␣␣2␣␣␣␣␣-234␣␣␣␣981233

You should verify that the effect of this is equivalent to

```
great = 23.44
blue  = -8.23899
movies= 2000
never = -234
fail  = 98123.3
```

10.1.4 Skipping Columns on the Input Line

To skip across (and ignore) columns in the input data, use a field descriptor of

```
nX
```

where n is the number of columns to skip over. For example, suppose we want to read a line that contains two integers. The first is in columns 10 to 15, and the second is in columns 55 to 60. So we want to skip over columns 1 to 9, read the first integer, then skip the next 39 columns, and read the second integer. The following would do this for us

```
      READ (5,102) mood, music
102   FORMAT (9X, I6, 39X, I6)
```

10.1.5 Skipping Lines of Input Data

As with **WRITE** statements, the slash may be used in **FORMAT** statements with **READ** statements to indicate that the current line is no longer required and that any further values should be read from the next line. For example, when the lines of data

```
␣␣123␣␣␣-12
␣-46104␣56
```

are read by the statements

```
      READ (5,130) more, lime, koops
130   FORMAT (I5 / I4, I6)
```

the variables **more**, **lime**, and **koops** are set to 123, –46, and 104056 respectively.

10.2 More Advanced Format Repeat Specifications

This applies to **FORMAT**s used for input or output. We have already seen that identical consecutive field descriptors may be repeated. For example,

```
140   FORMAT (1X, F10.3, F10.3, F10.3, I4, I3, I3, F10.3)
```

is equivalent to

```
140   FORMAT (1X, 3F10.3, I4, 2I3, F10.3)
```

Also, a whole group of field descriptors may be repeated by enclosing the group in parentheses and preceding the left parenthesis with an integer constant called the group repeat count, which indicates the number of times to interpret the enclosed group. If no group repeat count is specified, a group repeat count of one is assumed. For example,

```
150   FORMAT(1X, 2(F10.2/1X, I4), 2X, I5)
```

is equivalent to

```
150   FORMAT(1X, F10.2/1X, I4, F10.2/1X, I4, 2X, I5)
```

However, if the **FORMAT** is used for a **READ** or **WRITE** statement where there are more variables on the I/O list than there are field descriptors in the **FORMAT**, these two **FORMAT**s may not be treated as equivalent. See the next section for an explanation.

10.3 Interaction Between Read/Write and Format Statements

The number of values that will be read/written is determined solely by the number of variables in the I/O list in the **READ/WRITE** statement. The position of these values on the line is determined by the **FORMAT** statement. When a **READ** statement is executed, the next input line is read and thereafter additional lines are read only as the format specification demands (e.g. on encountering a slash in a **FORMAT** statement). When a **WRITE** statement is executed, a new line is started and thereafter additional lines are started only as the format specification demands (e.g. a slash). Except for the effects of the repeat count, the format specification is interpreted from left to right. To each I and F field descriptor in a **FORMAT** statement, there corresponds one variable in the I/O list in the **READ/WRITE** statement. To each literal (string in quotes) or X field descriptor, there is no corresponding variable in the I/O list. Whenever the format control encounters an I or F field descriptor in a **FORMAT** statement, it determines if there is a corresponding variable specified in the I/O list. If there is, the value of that variable is either read or written as specified by the field descriptor. If there is no corresponding variable, the **READ/WRITE** stops immediately. That is, the format continues until a corresponding variable cannot be supplied. For example, the statements

```
      a = 12.3
      b = 24.6
      c = -56.8
      WRITE (6,160) a,b
160   FORMAT(1X, 'This is a heading'/1X, 'A = ', F10.2/1X, 'B =',
     *       F10.2 / 1X, 'C = ', F10.2)
```

will write the lines

```
This is a heading
A =       12.30
B =       24.60
C =
```

10.3.1 Even More Advanced

If the format control reaches the last right parenthesis in a **FORMAT** statement, it tests to see if another variable is specified in the I/O list. If there are no further variables, then the I/O stops. However, if another variable is specified, a new line is started and format control reverts to the group repeat which is terminated by the second last right parenthesis, or if none exists, to the first left parenthesis. In the case of cycling back to a repeat group, the repeat count is used. For example, the statements

```
      ave   = 1.0
      best  = 2.0
      count = 3.0
      total = 4.0
      five  = 5.0
      WRITE (6,180) ave, best, count, total, five
180   FORMAT (1X, 'Fight = ', F6.2, 3X, 'Ring = ', F6.2, 3X, 'Bell')
```

would write out

```
Fight =   1.00   Ring =   2.00   Bell
Fight =   3.00   Ring =   4.00   Bell
Fight =   5.00   Ring =
```

Another example is

```
      WRITE (6,190) ave, best, count, total, five
190   FORMAT (1X, 'Fight'/2(1X, 'Ring = ', F6.2), 3X, 'Bell ', F6.2)
```

This would produce

```
Fight
Ring =   1.00 Ring =   2.00  Bell   3.00
Ring =   4.00 Ring =   5.00  Bell
```

The following output indicates what would be produced if the group repeat were expanded rather than being programmed as a repeat.

```
      WRITE (6,200) ave, best, count, total, five
200   FORMAT (1X, 'Fight' / 1X, 'Ring = ', F6.2, 1X, 'Ring = ',
     $        F6.2, 3X, 'Bell ', F6.2)
```

This would produce the following (note the extra line of `Fight`)

```
Fight
Ring =   1.00 Ring =   2.00   Bell   3.00
Fight
Ring =   4.00 Ring =   5.00   Bell
```

10.4 Reading/Writing Data Files

Rather than reading data from a terminal and printing results onto the terminal, Fortran can read and write data files directly by specifying a unit number other than 5 and 6. So, for example, the statement

```
READ(8,20) x,y,z
```

will read variables from the file connected to *Unit* 8. Files used for input are not restricted to 80 columns. The default line length is 132 characters. Files used for output, other than unit 6, are not treated as line printers, so they do not treat the first character on the output line in any special way. The default length of an output line is 132 columns. There are statements in Fortran to `OPEN` and `CLOSE` data files. The form of these statements varies between computers. Some computers expect all data files to be opened (or defined), while others only expect files with non-standard characteristics to be defined. As the treatment of data files is not standard, no further detail will be given here.

10.5 Exercises

Exercise 10A

Find the errors in the following Fortran statements and suggest ways in which they might be corrected.

```
1.        READ (5,100) i,j,a
    100   FORMAT(3I5)

2.        READ (5,110) i
          IF (j .GRT. i) WRITE (6,120) j
    110   FORMAT(I5.1)
    120   FORMAT('1',F6.2)

3.        DO 130 jjk = 1,10
          j = -5
          k = 0
          READ (5,130) a
          l = j/k+a
          IF (l .LT. j) STOP
    130   FORMAT(F6.2)
```

Exercise 10B

How many lines will be read?

```
1.        READ (5,160) a,i,b,j,crash,kreme
    160   FORMAT(F3.1,I5)

2.        READ (5,170) a,i,b,j,crash,kreme
    170   FORMAT(F3.1,I5,F3.1,I5)
```

Exercise 10C

How will the line of data

```
123456.12798.00012004567800.10
```

be read by the following statements?

```
1.        READ (5,180) i,wunda,wot,it,will,be
    180   FORMAT(I3,F7.0,F4.2,I4,F5.1,F4.2)

2.        READ (5,190) i,wunda,wot,it,will,be
    190   FORMAT(I5,F6.2,F3.1,I7,2F5.2)

3.        READ (5,200) i,wunda,wot,it,will,be
    200   FORMAT(I1,F3.1,F4.1,I3,F10.6,F3.2)
```

Chapter 11

Arrays

Consider the problem in which we read in data consisting of student marks in some subject. The problem is to read in the marks, and calculate the average mark. This may be done for N students using the following algorithm

1. Set the total to zero initially
2. Repeat for each of the N students
   (a) read in his/her mark
   (b) add it to the total
3. Calculate the average as the total divided by the number of students.

A Fortran program to do this is then

```
      READ (5,*) n
      sum = 0.0
      DO 20 m = 1,n
         READ (5,10) score
10       FORMAT (F6.2)
         sum = sum + score
20    CONTINUE
      ave = sum/n
```

Now consider a similar problem. We wish to calculate the average mark, and then print out those marks that are *greater than* the average. An algorithm (i.e. method) for solving this problem is

1. Read in the marks, summing them as they are read.
2. Calculate the average
3. Scan all the marks, to find and print out all marks greater than the calculated average.

To solve this problem we need to store *all* the exam marks when we read them, so that we will be able to use them a second time. One solution would be to read all the data twice. This would be inefficient as I/O is very slow compared to the rest of your program, and would also be inconvenient to the user, who would have to enter his data twice. Another solution you may propose is to store each of the marks in a

separate variable name. This approach suffers from two points. Firstly, what would happen if we had a large number of marks (say 9000), we would have that many variables! Secondly, this method would be very tedious, when it came to writing the 9000 **IF** statements to test if the variable value was greater than the average. This last statement implies that we wish to treat all the numbers in exactly the same way. The solution to our problem is to use an *array*. An array is a means of specifying a number of values that are to be stored under one variable name. In order to use an array, we must first declare two things

1. The variable name of the array
2. The size of the array. This is the maximum number of values that may be stored in the array.

For example, the statement

```
REAL grade(125)
```

defines an array called **grade** with space for 125 values, all of type real. These consecutive memory locations are called **grade(1)**, **grade(2)**, and so on up to **grade(125)**. Note that they each have the same name **grade**, and are distinguished by what we call a *subscript* — an integer enclosed in brackets. The individual memory locations are called *array elements*, and all 125 of them considered together are called an *array*. An array element can be used anywhere that a variable can, e.g. in an arithmetic expression, on the left hand side of an assignment statement or in a function reference. The real power in the use of arrays lies in the fact that we can refer to array elements, such as **grade(i)**. If i has the value 25, then **grade(i)** refers to **grade(25)**, and if i has the value 79 then it refers to **grade(79)**. In practice, we usually want to step along the array by making i take on the series of values which are successive subscripts into the array. This is done in a loop. For example, to read in N numbers into the array **grade**, we write

```
      DO 20 i = 1,n
         READ (5,10) grade(i)
10       FORMAT(F6.2)
20    CONTINUE
```

As the loop takes the next value of i, the next array element of grade is read from the next line of data. So now we are in a position to write the new program. The complete program is

```
C  Program to read in student grades, one per line, and
C  to calculate the average grade, then to print out all
C  those grades that are greater than this average.
C
C  The first line contains the number of grades (in cols 1-5)
C  The grades appear in columns 1-6
C  There is a maximum of 125 grades
C
      REAL   grade(125)
      READ (5,10) n
10    FORMAT(I5)
      sum = 0.0
```

```
C  Read in the grades and sum them
      DO 50 i = 1,n
         READ (5,40) grade(i)
40       FORMAT(F6.2)
         sum = sum+grade(i)
50    CONTINUE
C  Find the average
      ave = sum/n
C  Scan, printing all greater than the average
      DO 70 i = 1,n
         IF (grade(i) .GT. ave) WRITE(6,60) grade(i)
60       FORMAT (1X,F7.2)
70    CONTINUE
      STOP
      END
```

You will notice that although we do not know how many grades there are going to be — it is `n` — the size of the array **grade** is specified as 125. The rules of Fortran insist that the size of an array (as fixed in the declaration) be a positive integer constant, and 125 was chosen on the grounds that it will always be bigger than the number of grades used. Clearly, some judgement must be applied in choosing suitable sizes for arrays. An array that is declared unnecessarily large will waste memory space (or worse still, may not fit in memory!). An array that is declared too small, will result in errors occurring, as you are not allowed to reference array elements that are *out of bounds* of the declared array size. In many programs a test is made to check that you never try this. In the above example, we could improve the program by inserting the following lines afer the statement numbered 10

```
IF (n .GT. 125) THEN
   WRITE (6,*)  'Maximum of 125 grades allowed'
   STOP
END IF
```

11.1 Array Declaration

Arrays may be declared in one of three ways

1. by declaring the type of the array elements as well as the array size. For example,

   ```
   REAL marks(200), y(10)
   INTEGER x(10)
   ```

2. by using the default type which depends on the first letter of the array name. This is achieved with a **DIMENSION** statement, which has the form

   ```
   DIMENSION name(size), name(size), ...
   ```

   The declaration of **grade** in the previous section could also have been written as

   ```
   DIMENSION grade(125)
   ```

3. by declaring the array type and size in two separate statements. For example,

```
REAL marks
DIMENSION marks(200)
```

Notes :

1. Any number of **DIMENSION** and type declaration statements may appear in a program.
2. A name (variable or array) may appear only once in any declaration.
3. The size of an array must be an integer constant or an integer expression containing only constants.
4. **DIMENSION** statements and **REAL** and **INTEGER** type declaration statements must be placed before any executable statements.
5. All arrays used in the program must be declared, either in **DIMENSION** statements or **INTEGER** or **REAL** declarations.

11.1.1 Array Subscripts

When referring to a single array element, a subscript must always be specified. It is possible to use slightly more general subscripts than the simple integer constant or integer variable we have seen so far. Subscripts may be any integer expression, including function references and other subscripted variables. Most Fortrans will accept a fairly general form of integer expression as a subscript, but you should try to keep your subscript expression simple so it can be understood. If the version of Fortran that you use has a limited form of subscript, then, for example, just replace

```
top = grade(a+b)
```

with

```
ind = a+b
top = grade(ind)
```

So as well as referring to `grade(25)` and `grade(i)`, we can refer to `grade(i-10)`, which, if `i` has the value 25, refers to `grade(15)`; and also `grade(2*i+3)` which refers to `grade(53)`.

11.1.2 Examples of Subscripts

Whenever a subscript is used, any variables within the subscript expression must have been previously given a value, as the subscript expression is evaluated first and then this uniquely identifies a particular element within the array. It is the programmer's responsibility to ensure that the subscript expression evaluates to a number within the array's bounds. Some Fortrans will terminate with an error at execution time if an array subscript is out of bounds, and some do not, but merely give incorrect results. Provided that

1. the arrays mentioned have been declared,
2. the variables in the subscript expressions have been given a value at some prior point in the program, and

3. the subscript expression has a value between 1 and the dimensioned size of the array (inclusive),

then the following are valid array references

```
first (677)
hgiels (le)
a (max (lad,10))
hunter (lfs+1)
xmas (2*kanbra)
next (4*kraft - 3*kaas)
tooh (19*kat + mark(7))
```

Some examples of invalid subscripts (in the 1977 standard) are

Example	*Reason*
`dogs(bar)`	Real variable not allowed as a subscript
`katz(i+j+1.0)`	Real expression not allowed as a subscript

11.1.3 Example of Complex Subscript Use

Sometimes it is difficult to see why you would want to ever use a subscript that is more complicated than a simple integer variable. One example of this is as follows. Suppose that we have an array in which each set of three consecutive locations refers to rainfall figures: the minimum, the maximum, and the average. This means that if we have 25 areas, we would have 25 lots of 3 figures, and so would need an array of size 75. Suppose that variable `loc` indicates the area number that we are interested in. Now if we want the minimum rainfall figures for this area, it would be given by

```
rain ( (loc-1)*3 + 1 )
```

Value of LOC	*Element of RAIN*
1	1
2	4
3	7
4	10
etc.	etc.

11.2 Reading and Writing Arrays

Suppose that there are 10 integers on an input line, and that each integer takes 6 columns. Then the numbers may be read using the format

```
20    FORMAT(10I6)
```

The values could be read with

```
      INTEGER nums(10)
      READ (5,20) nums(1), nums(2), nums(3), ....,nums(10)
```

This is tedious, and seems to be made for a DO loop. So can we write this?

```
      INTEGER nums(10)
      DO 30 i = 1,10
         READ (5,20) nums(i)
20       FORMAT(10I6)
30    CONTINUE
```

The answer is *no*. The reason is that every READ statement will start reading a new line. The first time around the loop, the first number is read into nums(1) correctly. The second time around, a new line is read, and so the other nine values (that were on the first line) are lost. Clearly, some form of do-looping is required. Let us look at the correct piece of program to see how it is implemented.

```
      INTEGER nums(10)
      READ (5,20) (nums(i), i = 1,10)
20    FORMAT(10I6)
```

Here, instead of the usual input list on the READ statement, we have an input list in an *implied do-loop*. In this case the input list consists of the array element name nums(i). The input list is followed by a comma, which is followed by the implied do loop control

```
i = 1,10
```

and the whole lot is enclosed in brackets. It can be thought of as 'nums(i) for i going from 1 to 10'. There are really 10 variables on the input list, the variables nums(1) to nums(10). The implied do loop is merely a shorthand way of explicitly writing out the ten array elements. The list in the implied do loop may be part of a larger input/output list. For example, the statement

```
READ(5,60) p, bean, (nums(i), i = 1,10)
```

causes the first two numbers to be read into p and bean and then the next 10 into nums(1) to nums(10).

```
READ(5,70) (zot(i), i = 1,5), (nums(i), i = 1,10)
```

causes the first five numbers to be read into zot(1) to zot(5), and the next 10 numbers into nums(1) to nums(10). The input/output list in the implied do loop may contain several variables and array elements. For example,

```
READ(5,80) (zot(i), zowie(i), i = 1,5)
```

The first two numbers are read into zot(1) and zowie(1), the next two into zot(2) and zowie(2) and so on. The do-variable, initial, terminal and increment parameters follow the rules as for ordinary do loops. The following example will read in numbers into every second element of array spred. Notice that 10 numbers will be read, the first going into spred(1), and the last going into spred(19).

```
      INTEGER spred(20)
      READ (5,90) (spred(ind), ind = 1,20,2)
90    FORMAT(10I6)
```

The use of implied do loops is restricted to READ and WRITE statements only !!

11.3 Exception to the Rule

Normally, an executable statement can only act on one array element at a time, so an array name never appears without a subscript. The exception is that a whole array may be read or written by specifying the array name without subscripts. For example,

```
      INTEGER line(20)
      READ (5,*) line
```

will read 20 integers into `line(1)` to `line(20)`. Similarly,

```
      WRITE (6,10) line
10    FORMAT (1X, 20I5)
```

will write out all 20 elements. If only the first `n` (say) elements are required, then you must use an implied do loop, i.e.

```
      WRITE (6,10) (line(i), i = 1,n )
```

Note that although a variable number of array elements is written, the corresponding `FORMAT` cannot contain variables, but should allow for the maximum value of n. So the required `FORMAT` is again

```
10    FORMAT (1X, 20I5)
```

11.4 Common Errors with Arrays

A variable has a subscript but has not been declared as an array. A declared array name is referenced without specifying a subscript (except when the whole array is read or written). The size of an array specified in a declaration is not an integer constant. An array element is subscripted by a variable, which is either real, or does not currently have a defined value, or has a value less than 1 or greater than the maximum size specified in the declaration.

11.5 Exercises

All exercises involving arrays must include array declarations.

Exercise 11A

Find the mistakes in the following.

1.
```
      INTEGER j(20)
      DO 20 i = 1, 100
20    j(i) = 0
```

2.
```
      REAL  array(150)
      DO 50 k=2,47,2
50    array = array(k+3) + k
```

3.
```
      DIMENSION i(10),array(20)
      DO 70 i = 1, 20, 2
         array(i) = -3.
70       array(i-1) = real(i)
```

Exercise 11B

Write a program to calculate and print the average rainfall figures for each of 8 localities. Data consists of 8 lines of input (one line per locality). Each line of input has 12 numbers on it representing the rainfall for each of 12 months, for a locality. Use only one (one dimensional) array.

Exercise 11C

Identify the following declarations as being correct or not.

1. `INTEGER a(4), k(7)`
2. `REAL bad(12), rotten(17+12)`
3. `REAL good(k)`
4. `DIMENSION big(1000),small(3)`

Exercise 11D

Consider each pair of the following statements (not necessarily contiguous) to be in separate programs. For each pair, indicate any inconsistencies. This means, any errors that are *necessarily* errors, not merely potentially errors.

1.
```
DIMENSION x(5), l(10), a(15)
y = x + 65.0 / a(i)
```

2.
```
w = n(3) + i4 * 6
d = w**4
```

3.
```
delta = a(i) / x**8
start = delta**y - a(b)
```

4.
```
DIMENSION a(10), k(15), a(3), t(6)
t(15) = 2.0 * t(3) + k(5) * 3.14159
```

Exercise 11E

Write a series of statements to zero all locations in the array `b` from `b(1)` to `b(100)` inclusive.

Exercise 11F

Statistics are being kept on 20 southern Queensland national purple tailed hens. At the end of each month, a card is punched which contains the following information for each chicken.

Columns 1 to 2 contain an integer to be used for identification,
Columns 3 to 4 contain the number of eggs laid this month,
Columns 5 to 8 contain an integer which is the weight of feed consumed in grams,
Columns 9 to 12 contain an integer which is the weight of the bird in grams.

Write a program to read the 20 cards and print out for each hen,

1. identification number
2. number of eggs laid and difference from the average,
3. weight of feed consumed and difference from the average,
4. weight of the bird and difference from the average.

Chapter 12

Two-Dimensional Arrays

The arrays we have considered to date have been one-dimensional. We have been able to specify a unique array element by using only one subscript. In mathematical terms these arrays are vectors. We can also have arrays of two dimensions (and more, but we will not consider those in detail). Two-dimensional arrays may be considered as equivalent to matrices in mathematics. Suppose we were analyzing the results of a number of students who sat a number of exams. The results might be tabulated (by us) as

Student	*Chemistry A01*	*Forestry D31*	*Zoology A01*
Bloggs	66	72	51
Nurke	73	88	60
Eccles	50	71	75
Seagoon	24	12	51
Moriarty	77	79	62

For convenience, we identify students with student numbers and the subjects with subject numbers. So our table becomes

Student Num.	*Course 1*	*Course 2*	*Course 3*
1	66	72	51
2	73	88	60
3	50	71	75
4	24	12	51
5	77	79	62

Let us give this table a name, say call it **marks**. We may now refer to the mark obtained by student number 2 in subject number 3, by referring to the *second row* and the *third column* of the table **mark**, and we obtain the mark 60. We may represent this table in Fortran by declaring a two-dimensional array that has five rows (corresponding to the 5 student numbers) and three columns (corresponding to the 3 subject numbers). This may be done with the statements

```
INTEGER MARK(5,3)      or      DIMENSION MARK(5,3)
```

The reason for representing this as a two dimensional array, rather than a one dimensional array, is that all the Chemistry marks are in one column, and all of Eccles' marks are in one row. If we imagine a general version of this array where **nstuds** is the number of students (i.e. the number of rows in the table), and assuming that **mark** has been declared to have an appropriate size, then to find the total mark for subject 2, we would write the following

```
      itot = 0
      DO 5 i = 1, nstuds
         itot = itot + mark(i,2)
5     CONTINUE
```

Similarly, we could find the total marks for a particular student, say for Eccles, student number 3, with

```
      itot = 0
      DO 6 j = 1,nsubs
         itot = itot + mark(3,j)
6     CONTINUE
```

where **nsubs** is the number of subjects, i.e. the number of columns of the two dimensional array **mark**. Let us now generalise this a little further. We want a program that will read in the table of marks and store them in the array MARK and then will read requests for the total marks for either a particular student or a particular subject. We will indicate that we want a student total by reading a line with a 1 in column 1 and the student number we want in columns 3 – 5. If we want a subject total, then we will have a 0 in column 1, and the subject number in columns 3 – 5. Before we can attack this problem, we need to know how to read in a two-dimensional array.

12.1 Reading and Writing Two-Dimensional Arrays

A two-dimensional array is composed of a number of rows, each row being made up of a number of columns. One way of thinking about a two-dimensional array is to consider it as a number of one-dimensional arrays, each corresponding to a row, repeated for however many rows we have. The subscripts required to identify an element of the two-dimensional array are

1. a row subscript to identify which whole row to choose
2. a column subscript to identify which column to choose within the whole row specified by the row subscript.

The method of reading a two-dimensional array may be expressed in the following algorithm. Repeat for each row of the array

```
Read in values into a row vector.
```

Suppose we want to read numbers into a two-dimensional array of size 4 rows and 3 columns. There are 4 lines of data, each containing 3 numbers to go into one row of the array and each number taking 5 columns of a data line. First, look at how to read in any given row of the array, say row 1. This may be done with

```
      INTEGER nurgle(4,3)
      READ(5,110) (nurgle(i,j),j=1,3)
110   FORMAT (3I5)
```

To read in all 4 rows we could have

```
      READ(5,110) (nurgle(1,j),j=1,3)
      READ(5,110) (nurgle(2,j),j=1,3)
      READ(5,110) (nurgle(3,j),j=1,3)
      READ(5,110) (nurgle(4,j),j=1,3)
```

This may be abbreviated further by the use of a **DO** loop that will vary the row subscript from 1 to 4, and so giving

```
      INTEGER nurgle(4,3)
      DO 5 i = 1,4
         READ(5,110) (nurgle(i,j),j=1,3)
5     CONTINUE
110   FORMAT (3I5)
```

12.1.1 Nested Implied Do Loops

The syntax of an implied do loop is

```
(list of variables, control = start, finish, increment)
```

The list of variables may also contain a variable list in an implied do loop. Hence we can write

```
READ (5,109) ((nurgle(i,j), j = 1,3), i = 1,4)
```

This may be thought of as being '`nurgle(i,j)` with `j` going from 1 to 3 and `i` going from 1 to 4'. `j` is in the inner loop, and so changes most frequently (i.e. the inner loop is completed for each value of the outer loop). So this statement would read `nurgle(1,1)` then `nurgle(1,2)` then `nurgle(1,3)`, then `nurgle(2,1)` and so on. So the whole two-dimensional array may be read from one data line with

```
      INTEGER  nurgle(4,3)
      READ (5,101) ((nurgle(i,j), j = 1,3), i = 1,4)
101   FORMAT(12I5)
```

The implied do loop is merely a shorthand way of writing out all the array elements explicitly, so in the above **READ** statement there are really 12 variables on the input/output list. Suppose we wish to do the same thing, but with the data in a different format. If each line contains 3 numbers and there are 4 lines of data, each line is to be read into one row of **NURGLE**. We can do this by changing the **FORMAT** statement

```
      INTEGER  nurgle(4,3)
      READ(5,109) ((nurgle(i,j), j=1,3), i=1,4)
109   FORMAT(3I5)
```

This works, because when we get to the end of the **FORMAT** statement (which we will after reading three numbers), we start the format again and start reading another line. So a total of 4 lines is read. A much better way, because it is more straight forward, is to use an explicit do loop for the rows and an implied do loop for the columns, as in

```
      INTEGER  nurgle(4,3)
      DO 5 irow = 1, 4
         READ (5,110) (nurgle(irow,j), j=1,3)
5     CONTINUE
110   FORMAT(3I5)
```

12.2 Example

Let us now return to our exam mark example. We shall define the format of the data as being

1. Each line of data will contain the marks for a particular student in all subjects. There will be a total of 25 subjects, and the marks will be in I3 format.
2. There will be a maximum of 200 students, and the last line will be a 'dummy' student that has a negative mark, thus signalling the end of the data.

So to read in the marks we need

```
      INTEGER   mark(201,25)
      maxs = 200
C     Read in the student table

      DO 20 nrows = 1, maxs+1
         READ (5,10) (mark(nrows,j),j=1,25)
10       FORMAT(25I3)

C        Test for end of data
         IF (mark(nrows,1) .LT. 0) GO TO 30
20    CONTINUE

C     Only reach here if there are more than MAXS students
      WRITE (6,*) 'Too many students!!'
      STOP

C     Don't count the sentinel
30    nrows = nrows - 1

40    CONTINUE
C     continue with rest of program ...
```

So at this point in the program, we have set up the array of marks in the Fortran array called **mark**. Now we need to request particular student and subject totals. We have not specified how the number of requests will be indicated. Let us say that we should read requests until a type greater than 1 is read.

```
C     Read in a request
50    READ (5,60) itype, number
60    FORMAT(I1, 1X, I3)
C     Branch on the value of ITYPE
C     IF  itype = 0, form total for a given subject
C               = 1, form total for a given student
C               > 1, STOP
```

```
      IF (itype .GT. 1) STOP
      IF (itype .EQ. 0) GO TO 90

C     Find the total marks gained by the requested student
      itot = 0
      DO 70 j = 1, 25
         itot = itot + mark(number,j)
70    CONTINUE
      WRITE (6,80) number, itot
80    FORMAT(1X,'The total for student number ',I3,' is ',I5)
      GO TO 50
C     Form the total marks scored in the requested subject
90    itot = 0
      DO 100 i = 1, nrows
         itot = itot + mark(i,number)
100   CONTINUE
      WRITE (6,110) number, itot
110   FORMAT(1X,'The total for subject number ',I3,' is ',I5)
      GO TO 50
      END
```

Now let us put all the sections together. The data will be set up to read in the array as specified in the example above, and to request totals for student 2, and also for subject 1.

```
C
C     Author:      L. Landau
C     Date:        12 MAY 1976
C     Mods:        FEB 1981   L. Landau   To renumber statement numbers
C                  Aug 1984   K. Handel   Changed algorithm for reading
C                                          table
C     Program description:
C     This program will read in a table of student marks
C     Obtained in each of 25 exams.
C     The program will then read in requests for totals of either
C     (0)   total marks gained in a particular subject
C     (1)   total marks gained by a particular student
C     (>1)  stop program
C     Data description:
C     The student/subject table comes first in the data,
C     with each student taking one line,
C     and each subject taking 3 columns. if any marks are
C     omitted, then they will be treated as zero.
C     There will be a maximum of 200 students.
C     The end of the student data
C     is signified by a negative first subject mark.
C
C     Following the table of marks, there are requests for totals
C     of either student marks in a particular subject, or
C     the total marks for a particular student. these requests
C     take the form:
C
```

```
C       COLUMN        MEANING
C       ------        -------
C         1           type of request
c                     if 0 then total the student marks in the
c                     given subject
c                     if 1 then total the subject marks for the
c                     given student
c                     if greater than 1 then stop
c
c         3-5         the student number, or subject number.
C
      INTEGER  mark(201,25)
      maxs = 200
      maxsub = 25

C     Read in the student table

      DO 20 nrows = 1, maxs+1
         READ (5,10) (mark(nrows,j),j=1,25)
10       FORMAT(25I3)

C     Test for end of data
         IF (mark(nrows,1) .LT. 0) GO TO 30
20    CONTINUE

C     Only reach here if there are more than MAXS students
      WRITE (6,*) 'Too many students!!'
      STOP

C     Don't count the sentinel
30    nrows = nrows - 1

40    CONTINUE

C     Read in a request
      READ (5,50) itype, number
50    FORMAT(I1, 1X, I3)
C     Check that the request is in range
      IF (itype .GT. 1) STOP
      IF ((itype .EQ. 0 .AND. number .GT. maxsub) .OR.
     $   (itype .EQ. 1 .AND. number .GT. nrows)) THEN
         WRITE (6,60) itype, number
60       FORMAT(1X,'For a type ',I1,' request, number ',I3,
     $           ' is out of range....IGNORED')
         GO TO 40
      END IF
C     branch on the value of itype
C     itype = 0, form total for a given subject
C           = 1, form total for a given student
C
      IF (itype .EQ. 0) GO TO 90
```

```
C
C     Find the total marks gained by the requested student
      itot = 0
      DO 70 j = 1, maxsub
         itot = itot + mark(number,j)
70    CONTINUE
      WRITE(6,80) number,itot
80    FORMAT(1X,'The total for student number ',i3,' is ',i5)
      GO TO 50
C
C     Form the total marks scored in the requested subject
C
C
90    CONTINUE
      itot = 0
      DO 100 i = 1,nrows
         itot = itot + mark(i,number)
100   CONTINUE
      WRITE(6,110) number,itot
110   FORMAT(1X,' The total for subject number ',i3,' is ',i5)
      GO TO 50
      END
```

Then use the following data

```
66 72 51
73 88 60
50 71 75
24 12 51
77 79 62
-1
1  2
0  1
3
```

When the program was run on this data, it would print out

```
 The total for student number   2 is   221
 The total for subject number   1 is   290
```

This program is still very primitive. For example, it cannot handle the situation of students not sitting for exams. It provides only elementary output. What may be more interesting is to provide figures on who got more than 50%, or who the top student was. How would you make these changes? How much program re-design would it require?

12.3 Declaring Arrays with Lower Bounds

In the arrays we have done so far, array subscripts range from 1 to N, where N is the number of elements in the array. In 1977 Fortran, array subscripting has been made more flexible. The range of subscripts need not have a lower bound of

1. Subscripts may start at any integer, with the restriction that the upper bound must be greater than or equal to the lower bound. Unless the lower bound is 1, both the lower and upper bounds appear on the declaration, separated by a colon. The syntax for declaring a one-dimensional array where both lower and upper bounds need to be specified is

```
DIMENSION arrayname(lower-bound : upper-bound)
```

A type declaration may be used instead of **DIMENSION**. For example,

```
REAL number(0:9)
```

declares an array of 10 real numbers, with legal subscripts ranging from 0 to 9. So

`number(0)` = `1.0` is legal, but
`number(10)` = `1.0` is not legal.

If the lower bound and colon are omitted from the declaration, the default lower bound of 1 is assumed. Similarly, a two-dimensional array may be declared with

```
DIMENSION name(lower:upper, lower:upper)
```

For example,

```
DIMENSION graph(-5:5, -10:10), graph2(5, -10:10)
```

declares **graph** to have 11 rows, subscripted from −5 to +5, and 21 columns, subscripted from −10 to +10; and **graph2** to have 5 rows, subscripted from 1 to 5, and 21 columns, subscripted from −10 to +10.

12.4 Multi-dimensional Arrays

An array may have up to 7 dimensions. Multi-dimensional arrays may be declared as follows

```
INTEGER nums(60,132,10)
REAL    x(10,0:9,-5:4), y(-1:1,10:20)
```

nums is an array of integers with 60 rows and 132 columns, 10 layers deep. In memory, the order of storage is such that the first subscript (the row number) changes fastest and the last subscript changes slowest. **x** is a real 10 by 10 by 10 array. Legal array elements are

`X(1,0,-5), X(4,4,4), X(10,9,-1).`

y is a real 3 by 11 array. Legal array elements are

`Y(-1,10), Y(0,20), Y(1,15).`

Illegal elements are

`X(1,10,1), X(0,0,0), Y(1,1), Y(-1,5).`

12.5 Rainfall Example

Suppose that we have a table of rainfall figures for different areas over different years starting from 1950. So the table may look like

	Qld	*NSW*	*Vic*	*Tas*	*NT*	*ACT*	*SA*	*WA*
1950	103.5	99.3	89.6	121.6	41.2	88.8	84.5	77.4
1951	104.7	98.4	90.6	125.8	38.5	86.3	78.3	68.3
1952	109.8	101.6	92.4	132.6	39.5	85.4	79.5	72.1
1953	101.6	101.9	99.9	119.1	32.1	90.5	81.6	85.6
etc.								
1979	117.7	103.2	97.1	128.8	40.0	91.4	86.3	88.7

We will write a program to read these rainfall figures into an array and then extract rainfall figures for different areas over different years. The program to read the figures in will be generalized to accept more than just the known data, so an end-of-data marker will be a negative rainfall in the first area.

```
C     Maximum of 40 years  (starting at 1950)
C     Maximum of 10 areas
C
      REAL  rain( 1950:1990, 10)
      maxara = 10
      maxyrs = 40

C     Read in the number of areas
      READ (5,*) nareas
      IF (nareas .GT. maxara) THEN
         WRITE (6,30) maxara
30       FORMAT(1X, 'Too many areas, can only handle ',i3)
         STOP
      END IF
C     Read in rainfall figures. The end is indicated by
C     a negative rainfall for the first area.
      DO 50 iyr = 1950, 1990
         READ (5,40) (rain(iyr,loc), loc = 1,nareas)
40       FORMAT (16F5.0)
         IF (rain(iyr,1) .LT. 0) GO TO 80
50    CONTINUE

C     1990 is out of range
      WRITE (6,*) 'Too many years. Max is ', maxyrs
      STOP
C     Come here when all rain data is read
80    CONTINUE
      nyears = iyr - 1
```

Now the problem is to produce rainfall figures. Suppose that we want the average rainfall in Tasmania betwen 1960 and 1969 (inclusive).

```
      total = 0.0
      DO 90 iyear = 1960, 1969
         total = total + rain (iyear,4)
90    CONTINUE
      avrain = total/10
```

Generalising this a little, we want to solve the following problem . Read in 3 numbers, indicating

1. the start year (for example 1960)
2. the number of years to cover
3. the area number that we are interested in

and from this find the average rainfall in that area over the requested years.

```
C     Read in the input parameters: start year
C                                  number of years
C                                  location code
C     IBEGIN is the first and NYEARS is the last valid year.
      ibegin =  1950
      READ (5,*) ist, numyr, locat
C     check validity of requests
      IF (ist .LT. ibegin .OR. ist .GT. nyears) THEN
         WRITE (6,*) 'Starting year', ist, ' is out of range'
         STOP
      END IF

C     LAST is the last year requested
      last = ist + numyr - 1
      IF (last .GT. nyears) THEN
         WRITE (6,*) 'Final year', last, ' is out of range'
         STOP
      END IF

      IF (locat .LT. 1 .OR. locat .GT. nareas) THEN
         WRITE (6,*) 'Location', locat, ' is out of range'
         STOP
      END IF

C     Now we have validated the input so do the calculation
C
      total  = 0.0
      DO 160 iyear = ist, last
         total = total + rain (iyear,locat)
160   CONTINUE
      avrain = total/numyr
      WRITE (6,170) locat, avrain
170   FORMAT(1X,'The average rainfall in area ',I3,
     $       ' is ',F8.2,' centimetres')
      STOP
      END
```

12.6 Exercises

Exercise 12A

Write a series of statements to add the corresponding elements of two m by n arrays A and B and store the result as an m by n array C. Assume that the maximum values of m and n will be 14 and 10 respectively.

Exercise 12B

Write a program to generate the numbers 1 to 30 in a one dimensional array and then print this array so that there is a heading before the numbers, and then the numbers on one line. Print the numbers again, six per line, with a line number on each line. The output should be

```
THE NUMBERS ARE
 1  2  3  4  5  6  7  8  9  . . .  30
   1     1     2     3     4     5     6
   2     7     8     9    10    11    12
   3    13    14    15    16    17    18
   4    19    20    21    22    23    24
   5    25    26    27    28    29    30
```

Exercise 12C

Repeat exercise 12B, but using a two dimensional array with 10 rows and 3 columns. The output should be exactly the same. Fill up the 10 by 3 array by rows rather than by columns.

Exercise 12D

Re-write your answer to exercise 11B, this time also printing out the entire rainfall figures for the month with the highest average. Use only one two-dimensional array.

Exercise 12E

For each of 8 areas, read a monthly rainfall figure (one real number per line) for each of 12 months. Find the average monthly rainfall for each area, and the average monthly rainfall over all areas. Write an algorithm first. Note: Arrays are not necessary.

Exercise 12F

The coordinates of points in an n dimensional space are written in a data file, one set per line. The value of n is written on the first line, by itself. Write a program to read these values and find the distance d of each point from the origin.

$$d = \sqrt{x1^2 + x2^2 + \cdots + xn^2}$$

Assume that n will not be greater than 25.

Chapter 13

Arrays as Subroutine Parameters

Now we shall put arrays and subroutines together.

13.1 Actual Parameters

The actual parameter list on a subroutine call may include array names and single array elements as well as constants, variables, expressions and function names. If a whole array, or a portion of an array is to be used by the subroutine, then the parameter list should include the array name without any subscript or dimension. If the value of one array element is passed to a subroutine, then the subscripted element name is put in the actual parameter list, and will correspond to a variable in the dummy parameter list of the subroutine. The use of array names in actual parameter lists is similar to the use of variables, in that values may be passed to the subroutine and new values may be passed back from the subroutine. The use of array elements in actual parameter lists is similar to the use of constants and expressions, in that values may be passed into the subroutine, but the corresponding dummy parameters may not be given new values in the subroutine, i.e. new values may not be passed back from the subroutine.

13.2 Dummy Parameters

The dummy parameter list on a subroutine header may include array names as well as variable names. No subscript is allowed with an array name. In other words, arrays are not dimensioned in the subroutine header. They are dimensioned in the declaration statements that follow the header. For example,

```
SUBROUTINE size(big,small,n)
REAL big(100), small(50)
```

passes the real arrays `big` and `small` and the integer variable `n` between the calling program and the subroutine. The arrays must be declared as arrays in both the calling program and the subroutine. For example, the program which calls the above subroutine might be

```
      REAL array1(100),array2(50)
        ...
      CALL size(array1,array2,n)
```

The parameters may be given values before the subroutine is called or may be given values in the subroutine and the answers passed back to the main program. Remember that a variable in the dummy parameter list is only an alias name for the corresponding actual parameter value. In other words, it is a pointer to the actual value. Similarly, an array name in a dummy parameter list is a pointer to its corresponding actual parameter, and a reference to an array element in the subroutine is actually a reference to the corresponding array element in the calling program. If you forget to declare the actual parameter as an array in the calling program, and the subroutine references the 5th element of the array, then it will access a memory location which is 4 locations past the variable which is the actual parameter. So, only the location of the first element of an array is in fact passed to/from the subroutine header.

13.2.1 Example

Read 10 values into array **x**. Call subroutine **eval** to calculate 10 values for **y** depending on the values in **x**.

```
      REAL x(10), y(10)
      READ(5,*) (x(i), i=1,10)
      CALL eval(x,y)
      WRITE(6,10) (x(i),y(i), i=1,10)
10    FORMAT(1X,2F8.3)
      STOP
      END

      SUBROUTINE eval(x,y)
      REAL x(10), y(10)
      y(1) = x(1)
      DO 10 i = 2,10
          y(i) = x(i)**2 - x(i-1)*3
10    CONTINUE
      RETURN
      END
```

13.3 Adjustable Array Dimensions

The subroutine in the example above will work for any two real arrays of exactly size 10. We want to be able to write a more general subroutine which will work on arrays of any size. For example, a useful subroutine is one which sorts the elements of an array into ascending order. We want this subroutine to work on any array, not just one of a particular size. This can be done in a subroutine by giving the array adjustable, or variable, dimensions. Adjustable dimensions only apply to arrays in the parameter lists of subprograms (subroutines and functions) because these are the only arrays that do not have space set aside for them at compile time. Arrays in the main program, and arrays in subroutines other than those in the dummy parameter lists are allocated a fixed amount of space at compile time and so the

declarations must specify fixed dimensions. For an array to be adjustable, both the array name and the variable whose value is the adjustable size must be in the parameter list. For example, the statements

```
      SUBROUTINE eval(x,y,n)
      REAL x(n), y(n)
```

declare **x** and **y** to be adjustable arrays each of dimension **n**, where **n** must be given a value in the calling program prior to the subroutine call. The size of the arrays is unknown when the subroutine is written, and may vary from one call to the next, but at each call, the size (i.e. **n**) must be known. The body of the subroutine must also be generalized to use **n** rather than 10 in the above example. So the above example becomes

```
      REAL x(10), y(10)
      READ(5,*) (x(i), i = 1,10)
      CALL eval(x,y,10)
      WRITE(6,10) (x(i),y(i), i = 1,10)
10    FORMAT(1X,2F8.3)
      STOP
      END

      SUBROUTINE eval(x,y,n)
      REAL x(n), y(n)
      y(1) = x(1)
      DO 10 i = 2,N
          y(i) = x(i)**2 - x(i-1)*3
10    CONTINUE
      RETURN
      END
```

Unfortunately, there is still the restriction that the arrays must be of the specified type. So a subroutine may be made general enough to (say) sort the numbers in any size array, but it can only sort an array of reals *or* an array of integers. There is no way around this restriction.

13.3.1 Example

Suppose we want to write a subroutine that will calculate the sum and average of the elements of any one-dimensional array. First choose a name, say, **calsum**, and decide on the dummy parameters. They will include the number of elements of the array, say **n**, which is an integer. Also we need the name of the array, say **a**, which we will say is of type real, and then the sum, say **sum**, which again is real, and lastly the average **ave** which is also real. This produces the subroutine header line of

```
      SUBROUTINE calsum ( a,n,sum,ave )
```

Second, write the body of the subroutine to perform the desired operations on the dummy parameters

```
      SUBROUTINE calsum ( a,n,sum,ave )
C
C   Author:    L. Landau
C   Date:      SEPT 1977
```

```
C   Purpose:
C      To calculate the sum and the average of the first n
c      elements of an array
C
C   Parameter description
C   ---------------------
C
C    Any parameters with an * to their left will be altered
c    in the subroutine
C
C     A         the name of the array              (real)
C     N         the size of the array 'a'          (integer)
C  *  SUM       the total of 'n' elts of 'a'       (real)
C  *  AVE       the average of the first 'n'       (real)
C               elements of the array 'a'

      REAL  a(n)
      sum = 0.0
      DO 5 i = 1,n
         sum = sum + a(i)
5     CONTINUE
      ave = sum/n
      RETURN
      END
```

If the subroutine were to include a fixed format `WRITE`, then the corresponding `FORMAT` would have to put a fixed number of values on each line. For example,

```
      WRITE (6,10) ( a(i), i = 1,n )
10    FORMAT (1X, 5F10.3)
```

The following statement is illegal

```
10    FORMAT (1X, N(F10.3) )
```

13.4 Adjustable Two-Dimensional Arrays

As an array in memory is physically a group of consecutive locations, it is easy to see the correspondence between actual and dummy array elements in a one-dimensional array. However, if a two-dimensional array is to be passed to a subroutine and only a portion of the array is to be used, then the required locations are not all consecutive. For example, suppose an actual array `a` has 20 rows and 10 columns, and only the first 15 rows and 5 columns are to be used by a subroutine. Then if the subroutine were to declare the array to be `m` by `n` where `m` is 15 and `n` is 5, this would imply that `a(1,2)` comes directly after `a(m,1)` in memory (remembering that arrays are stored by column). However, in the actual array, `a(15,1)` is followed by `a(16,1)` in memory. So the two arrays do not match. Hence the full size of a two-dimensional array must be passed to a subroutine. This example would require

```
      REAL a(20,10)
      CALL sub(a, 20,10, 15,5)
        ...
```

```
      END

      SUBROUTINE sub(a,nrows,ncols,m,n)
      REAL a(nrows,ncols)
        ...
      use the first M rows and N columns
        ...
      END
```

13.5 Example of Subroutine Use

This example shows how to divide a problem into sub-problems. Each sub-problem is handled in a subroutine.

Problem — We have a record collection and wish to maintain a register of items, so that it may be easily updated, listed and maybe in the future, resorted. At the moment, we can only process numerical data, so the record title, etc. will have to be coded. The data to be kept may be such things as

1. record number
2. location code (shelf number, borrowed, missing)
3. music type
4. date obtained
5. playing time (minutes and seconds)

The program should

1. Read in all the data.
2. Edit the data, and print out any formatting errors.
3. List the data, printing 40 entries per page with a heading and page number on the top of each page.

The main program may look something like this

```
c     comments describing the codes used and input data etc.
c
c     subroutines used :
c
c     read      this reads in data for one record and returns
c               information in an array
c
c     edit      checks the information read in and returns a
c               parameter to indicate if there was an error.  it is
c               responsible for printing error messages too.
c
c     head      prints a heading on the top of a page.
c               also increments a page counter, and resets
c               the line count to zero.
```

```
c
c     output   writes out a line, increments line count
c
      INTEGER  music (10)
      LOGICAL ok
      lincnt = 0
      ngood = 0
      ipage = 0
      CALL head (ipage,lincnt)
c
c     read in a music record. end of file is indicated by a
c     negative record identifier
c
5     CONTINUE
      CALL read(music,10)
      IF (music(1) .LT. 0) GO TO 99
c
c     check validity of the entry
c     and count the number of invalids
c
      CALL edit(music,10,ok,lincnt)
      IF ( ok ) ngood = ngood + 1
      CALL output(music,10,lincnt)
      IF (lincnt .GT. 40) CALL head (ipage,lincnt)
      GO TO 5
c
c     end of file found
c
99    CONTINUE
      WRITE (6,100) ngood
100   FORMAT('1', 'There were ', i3, ' valid entries')
      STOP
      END
```

Each subroutine would be described by a block of comments at its head saying

1. what parameters it uses
2. what values are returned
3. what the subroutine does

The main program does not do very much but call a number of subroutines, but it is easy to follow and provides an ideal starting point for understanding or changing the program.

In this example, **read** is used as a subroutine name. Fortran does not have protected keywords, so any word may be used as a variable, array or subroutine name. However, all names in a program unit must be unique, so a variable called **max**, for example, cannot be used in the same program unit as the supplied **max** function.

13.6 Exercises

Exercise 13A

Given a subroutine header and declarations of

```
SUBROUTINE jedda(dg,max,log,strad)
REAL  dg(max,max), log(max,10)
```

and the following declarations in the calling program

```
DIMENSION ct(44,44), sunny(44,44), kash(10,10)
```

which of the following calls are legal ?

1.
```
CALL jedda(ct,18,sunny,14.8)
```

2.
```
isz = 30
brok= 98.4
CALL jedda(sunny,isz+3,ct,brok+isz)
```

3.
```
CALL jedda(kash,8,ct,9.1)
```

4.
```
CALL jedda(ct,6,sunny)
```

5.
```
CALL jedda(14,ct,sunny,18.9)
```

6.
```
CALL jedda(ct(13,4),10,sunny(3,3),ct(2,2))
```

7.
```
CALL jedda(ct,45,sunny,ct(2,3))
```

Exercise 13B

ja is a one dimensional array with 50 elements. Write a subroutine to compute the average of the first **n** elements and a count of the number of these elements that are zero. Call the subprogram **avernz(ja,n,aver,nz)**. Write a main program to read **n**, then read **n** elements into **ja**, call **avernz** and then print the answers.

Exercise 13C

Write a subroutine that reads in a two dimensional array, using adjustable dimensions.

Exercise 13D

Write a complete program, consisting of a main program and three subroutines, which will amend a list of numbers as follows. There must be an even number of numbers. Every odd-numbered element in the list (i.e. the first, third, fifth, etc. elements) is to be replaced by the sum of itself and the next element, and every even-numbered element is to be replaced by the absolute difference between itself and the previous element. That is, successive elements will then contain sums and differences. The main program should read the number of numbers and check that it is even. It then should call the first subroutine to read the numbers to be amended. The main program should then call the second subroutine to amend the numbers as described above. The third subroutine should be called to print the original numbers and then be called again to print the amended numbers. All input should be in fixed field format and all output should be in a presentable form with suitable headings.

Chapter 14

Functions

Functions have already been introduced in a previous chapter. So far the functions that have been used have been supplied Fortran functions such as `MOD`, `ABS`, `SIN` etc. Each function has a specific name, a list of actual parameters and a type of result (i.e. integer or real). Functions, like subroutines, are subprograms. Functions are used directly within arithmetic expressions, unlike subroutines which require a `CALL` statement to invoke them. The form of a function is very similar to that of a subroutine. The form of a function is

```
FUNCTION  name  (dummy parameters)
  ...
body of the function
  ...
END
```

14.1 Function Header

The first line is the function header, which identifies the lines that follow as being a function. The function header contains

The Name — This is the name of the function (supplied Fortran function names are `MOD`, `ABS`, etc.) and it follows the rules for variable names. The *type* of the name (i.e. integer or real) identifies the type of result that the function will return. So a function whose name is of type integer will return an integer result, similarly a real name would mean that the function will return a real result. The result of a function is that value which is assigned to the name of the function from within the body of the function. That is, in the function the function name is treated as a variable. It is assigned a value (within the function), and this is the value which is returned to the calling program.

Dummy Parameters — These are the same as specified in subroutines. When the function is used, it will have corresponding actual parameters. If there are no parameters then the brackets may be omitted from the function header, but they should be included in the function reference (so that the compiler can distinguish it from a variable reference).

14.1.1 Body of a Function

The last line of a function must be

```
      END
```

The body of a function contains

1. At least one **RETURN** statement.

2. At least one assignment of a value to the name of the function.

The **RETURN** statement is used in the same way as it is in a subroutine: it specifies that execution of the calling program is to continue on from the point at which the function was referenced (the calling program may be a main program, a subroutine or a function). The assignment of a value to the function name is done so that a value may be returned from the function. This assignment must be by either

1. the function name appearing on the left hand side of an assignment statement, or

2. the function name appearing in a **READ** statement.

14.2 Example of a Function

Suppose that we want to find the average of the first n numbers in a real array. We will write **FUNCTION aver** to return this answer. This average will be used to write out a message. If the average is greater than 50.0 then write out that the average is good; otherwise write out that the average is low. The function, with some of the calling program, is

```
      REAL  values (100)
      1. read in a value for N
      2. test to ensure that N is less than or equal to 100
      3. read values into the array
      IF (aver(values,n) .GT. 50) THEN
          WRITE (6,*) 'The average is good'
      ELSE
          WRITE (6,*) 'The average is low'
      END IF
      STOP
      END

      FUNCTION aver(arr,num)
      REAL arr(num)

C     Calculate the average of the first NUM elements of ARR
      total = 0.0
      DO 25 i = 1,num
         total = total + arr(i)
25    CONTINUE
      aver = total/num
      RETURN
      END
```

14.3 Explicit Type Declarations for Functions

The default type of the result of a function is given by the type of its name. Just as the default type of a variable name can be overridden by an explicit declaration, so we may do the same with a function, by specifying its explicit type on the function header line as

```
type FUNCTION name (dummy parameters)
```

For example

```
REAL FUNCTION ns(a,i)
```

would return a real number even though the name **ns** is an integer name. It is as if the name of the function (in this case **ns**) appears in a declaration. The function type must be the same in the calling program and the function, so if the default type is not used then the function name must also appear in a declaration in each program (or subprogram) where it is referenced. In the above example, the calling program would need the declaration

```
REAL ns
```

In the function, the function type cannot appear in a separate declaration, but must be put on the function header.

14.3.1 Example of a Logical Function

As there is no default type for a LOGICAL function, you must explicitly declare the type of the function name in the program (or subprogram) that references the function. For example, suppose that we have a program to read in an array of numbers that represents the salaries of people. Further suppose that these salaries are sorted into ascending order, but just to ensure this we have a LOGICAL function called **edit** to test for this, and also to test that no salary is less than 100.0 nor greater than 100,000.0 . In the data, salaries appear one per line, with the last one being negative. The values are entered in columns 1–10.

```
C   Author:   L. Landau
C   Date:     October 1979
C   Input Description:
C         Salaries are entered one per line as real numbers, in
c         columns 1-10.
c         end of data is signalled by a salary less than zero
C
c   subprograms used:
c         edit    logical function to test validity of data

      LOGICAL edit
      REAL salary(1001)
      maxnum = 1000
      irow = 1
c
c   read in salaries
c
10    CONTINUE
```

```
      READ(5,20) salary(irow)
20    FORMAT(F10.0)
      IF (salary(irow) .LT. 0.0) GO TO 40
      irow = irow + 1
      IF (irow .LE. maxnum+1) GO TO 10
      WRITE(6,30) maxnum
30    FORMAT(1X,'Too many salaries, can only handle ',i4)
      STOP
c
c   come here when end of data is found
c
40    CONTINUE
      irow = irow - 1
      IF (edit(salary,irow)) GO TO 60
      WRITE(6,50)
50    FORMAT(1X,'Salaries not sorted or out of range')
      STOP
60    CONTINUE
      .
      .
      THE REST OF THE MAIN PROGRAM WOULD COME HERE
      .
      .
      END

      LOGICAL FUNCTION edit(salary,n)
c
c   author:  l. landau
c   date:    oct 1979
c   purpose:
c     to check that the array is sorted into ascending order and
c     that all the numbers lie in the range of 100.0 to 100,000.0
c
c   parameter description:
c
c       salary      ....  a real array of salaries
c       n           ....  the number of salaries in the array
c
c value returned by the function:
c
c      if the data is ok  then return .true.
c      if the data is nbg then return .false.
c
      REAL salary(n)
      IF ( salary(1) .LT. 100.0 .OR.
             salary(n) .GT. 100000.0) GO TO 90

      DO 5 i = 2,n
        IF ( salary(i-1) .GT. salary(i) ) GO TO 90
5     CONTINUE
c
```

```
c     all ok so return true
c
      edit = .TRUE.
      RETURN
c
c     come here if errors found
c
90    CONTINUE
      edit = .FALSE.
      RETURN
      END
```

14.4 Functions Vs Subroutines

A function is usually used instead of a subroutine when one value is to be returned to the calling program (although other values may be returned in the parameter list). For example, calculating the area of a triangle from its sides is more suited to a function than to a subroutine. The technique used in writing the function is basically the same as for the subroutine.

1. Choose a *name* and *type* for the function (say **REAL FUNCTION area**) and decide on its dummy parameters (the real quantities **a**, **b**, and **c** representing the sides). Thus we arrive at the heading

   ```
   REAL FUNCTION area (a,b,c)
   ```

   or just

   ```
   FUNCTION area (a,b,c)
   ```

2. Write the body to calculate the appropriate value from the dummy parameters, and assign that value to the name of the function. This is quite simply done here by calculating **s**, and then **area** and then returning.

   ```
   REAL FUNCTION area(a, b, c)
   s = 0.5*(a + b + c)
   area = SQRT( s*(s-a)*(s-b)*(s-c))
   RETURN
   END
   ```

This is basically the same as the subroutine to calculate the area of a triangle in chapter 8, but the function is preferable because the purpose of the code is to return one specific value which may be used in an expression in the calling program. The calling program might reference the function with

```
diff = ABS ( area(a1,a2,a3) - area(b1,b2,b3) )
WRITE (6,*) 'The difference between the triangle areas is',diff
```

where **a1**, **a2**, **a3** are the sides of one triangle and **b1**, **b2**, **b3** are the sides of another.

14.5 Why Use Subprograms?

Subroutines and functions may be used as self-contained building blocks to make a program easier to write. If a problem can be divided into sub-problems that can be solved easily, then the Fortran program which solves it can be designed in a similar way. This is the divide and conquer strategy of solving problems. Further, since a subprogram may exist in isolation from a main program, it may be tested independently also, and when it has been proved to work, it may be combined with the rest of the program. If a subprogram is general, it may be used in a variety of situations, thus saving much repetitive effort on the part of the programmer. Programs that use subprograms are easier to follow and easier to maintain, two of the desirable goals that make a program 'better'. Most computer installations maintain a library of subprograms which may be used to solve common problems such as sorting, solving sets of linear equations, returning the date and time, statistical analysis subroutines, and many more. These subprograms have been written so that they are general, and also robust. This means that they are likely to return an error message if the user puts unacceptable data into them, rather than crashing the program with no clues to the mistake. You should write robust programs too!

14.5.1 Common Errors and Points to Note

There must be the same number of parameters in the use of a subprogram as there are in the definition. Variable types in both parameter lists must match, i.e. if a parameter is an integer in the definition, then an integer must be supplied as the actual parameter. If an array name is used as a parameter, then it must be dimensioned in the calling program and the corresponding parameter in the definition must also be dimensioned. Both of these dimensions must be identical, unless adjustable dimensions are being used. In this case, the array must be defined in the calling program with a fixed dimension, and the values of the dimensions must be supplied as parameters to the subprogram. Only arrays in the parameter list can have adjustable dimensions. The name of a function must appear at least once in the body of the function on the left hand side of an assignment statement or in a `READ` statement. Its name must be unique, i.e. no other variable in the calling program can have the same name as the function. The name of a subroutine must not appear in any statement in the body of the subroutine (except in the header) and must not appear as a variable name in the calling program. That is, the name must be unique. If the definition of a subprogram changes the value of a variable in the parameter list, then a constant cannot be used as the corresponding actual parameter, or the value of the constant may be changed. A `RETURN` statement or a logical `IF` statement containing a `RETURN` statement must not be the last statement of a `DO` loop. A `CALL` statement or a logical `IF` statement containing a `CALL` statement must not be the last statement of a `DO` loop.

14.6 Exercises

Exercise 14A

Write a real function to calculate the area of a circle of radius r.

$$area = \pi \times r^2$$

where $\pi = 3.14159265$

Exercise 14B

Define an real function to compute

$$f(x) = x^2 + \sqrt{1 + 2x + 3x^2}$$

Then use the function to compute

1.

$$a = \frac{6.9 + y}{y^2 + \sqrt{1 + 2y + 3y^2}}$$

2.

$$b = \frac{2.1z + z^4}{z^2 + \sqrt{1 + 2z + 3z^2}}$$

3.

$$c = \frac{\sin y}{y^4 + \sqrt{1 + 2y^2 + 3y^4}}$$

4.

$$d = \frac{1}{\sin^2 y + \sqrt{1 + 2\sin y + 3\sin^2 y}}$$

Exercise 14C

Write a function subprogram, with two parameters r and p, that calculates the area of an equilateral triangle of side r when $p = 1$, a square of side r when $p = 2$, a circle of radius r when $p = 3$.

Exercise 14D

Write a function subprogram for which the parameter list contains `a`, `m`, and `n`, where `a` is an array name, and `m` and `n` are the numbers of rows and columns respectively. The function value is to be the sum of the absolute values of all the elements in the array. The dimensions are to be adjustable.

Exercise 14E

Write a function subprogram that searches a one dimensional array and returns the largest value.

Exercise 14F

Write a function that will count the number of zeros in a two dimensional integer array.

Exercise 14G

Write a function to find the factorial of an integer which is supplied to the function. Return the answer (do not print it in the function).

Exercise 14H

Rewrite your answer to exercise 8F, using functions instead of subroutines.

Chapter 15

Character Variables

15.1 Declaration

In 1977 a new data type was introduced, called type *character*, which is used to store a number of characters. The form of declaration of character variables is

```
CHARACTER*l  variable list
```

The 'l' indicates the length, or number of characters that may be stored in each of the variables on the list. In addition to this, or instead of this, variables on the list may be followed by an asterisk and a length specification. For example

```
Line 1      CHARACTER*20 name, addr
Line 2      CHARACTER name*20, addr*20
Line 3      CHARACTER*5 axle*10, mine, rob*100
```

Line 1 indicates that the two variables **name** and **addr** are character variables, and can each store 20 characters. Line 2 has the same effect as line 1, but does it by including individual length specifications. Line 3 declares that any variables that do not have their own length specifications will be able to store 5 characters each. So, **axle** can store 10 characters, **mine** can store 5 and **rob** can store 100. There is no default length for a character variable, so the following declaration is illegal

```
CHARACTER name
```

15.2 Character Arrays

Character arrays are similar to integer or real arrays except that they contain (a fixed number of) characters in each element of the array. Character arrays can be dimensioned in a dimension statement or within the character declaration. For example, the following declarations are identical, and they each declare `trial` to be a character array of size 100 and `x` to be a character array of size 10. Each element of `trial` can store 48 characters, and each element of `x` can store 2 characters.

```
CHARACTER      trial*48,x*2
DIMENSION      trial(100),x(10)
CHARACTER*48   trial(100),x(10)*2
CHARACTER      trial*48(100),x*2(10)
CHARACTER*48   trial(100)
CHARACTER*2    x(10)
```

The latter form is perhaps the most consistent with other type declarations.

15.3 Character Constants

A character constant, also called a literal or text constant, is any string of characters enclosed within single quotes. If a single quote is to appear within the character constant, then it must be immediately followed by another single quote. So far, we have been using character constants in headings in **WRITE** statements. Now they can also appear within character expressions, analagous to integers and reals appearing within arithmetic expressions. If a character constant appears on the right hand side of an assignment statement, then its value is placed in the character variable on the left. The constant is truncated if it is too long, or is placed left justifed in the variable, with blanks filling the rightmost characters if the constant is too short. For example

```
CHARACTER man*8, fuss*12
man  = 'abcd'
fuss = 'The boy''s job is difficult'
```

The value stored in `man` is — abcd␣␣␣␣
The value stored in `fuss` is — `The boy's jo`

A ␣ indicates a blank.

15.4 Substrings

A substring is a portion of a string. Reference is made to substrings with

`VAR(e1:e2)` or `ARRAY(subscripts)(e1:e2)`

where `VAR` is a character variable, `ARRAY(subscripts)` is an element of an array of type character, and `e1` and `e2` are integer expressions. The value of `e1` specifies the leftmost character position of the substring. The value of `e2` specifies the rightmost character position of the substring. If len is the length of the character variable, then

$1 \leq e1 \leq e2 \leq len$

If `e1` is omitted, a value of 1 is implied. If `e2` is omitted, a value of len is implied.

15.5 Examples of Substrings

1.
```
CHARACTER*20 state
CHARACTER*10 name
state = 'Western Australia'
name  = state (9:17)
state(1:7) = state (9:15)
```

After this,

`name` is `'Australia'`
`state` is `'Austral Australia'`

2. Assuming the original declarations above, the statements

```
state = 'abcdefghijklmnopqrstuvwxyz'
name  = state(5:20)
```

will store

'abcdefghijklmnopqrst' in state and
'efghijklmn' in name

3. Find all the blanks in a character variable called line

```
      CHARACTER*80 line
      READ (5,*) line
      DO 10 i = 1,80
      IF (line(i:i) .EQ. ' ') WRITE(6,*) 'Blank in position ',i
10    CONTINUE
```

line(i:i) looks a bit like an array element, but it is one character position in a character variable.

4.
```
      CHARACTER*4 c(2,2),c1
      c1 = 'ABCD'
      c(1,1)(3:4) = c1(1:2)
      c(1,1)(1:2) = c1(3:4)
```

puts 'cdab' into c(1,1).

15.6 Reading and Writing Characters

15.6.1 Free Format

Character variables may be read and written using free format. In writing, the resultant output takes exactly as many columns as the size of the character variable. That is, it is not surrounded by blanks. In reading, the data is presented as a character constant, namely, a string of characters *enclosed in single quotes !*

15.6.2 Fixed Format

Input and output of character variables and arrays is handled by the A field descriptor which has the form

```
Aw
```

where w indicates the width of the input or output field. If the w is omitted then the declared length of the character variable that is being read/written is assumed to be the width of the field. If w is specified and is different to the length of the variable then on input

1. if w < length then w characters are placed left justified with blank fill, in the variable;
2. if w > length then the rightmost 'length' characters are taken from the input field;

and on output :

1. if **w** < length then the leftmost **w** characters are output;
2. if **w** > length then 'length' characters are written, right justified in a field of length **w**.

15.6.3 Example of Reading and Writing

Consider the following program

```
      CHARACTER*10 code, locat
      CHARACTER*4  med, lost
      READ (5,10) code, locat, med, lost
10    FORMAT (A5,A12,A3,A)
      WRITE (6,20) code, locat, med, lost
20    FORMAT (1X,A,A,2A6)
```

If the input data is

```
abcdefghijklmnopqrstuvwxyz
```

then the result of reading is

Variable	*Value*
code	abcde␣␣␣␣␣
locat	hijklmnopq
med	rst␣
lost	uvwx

and the output is

abcde␣␣␣␣␣hijklmnopq␣␣rst␣␣␣uvwx

15.7 Character Operators

The only character operator is the concatenation operator (//)

```
expr1 // expr2
```

where **expr1** and **expr2** are character expressions. The value of the above expression is a character value that is the first expression immediately followed by the second. For example

1.
```
CHARACTER a*4,b*8
a = 'abcd'
b = a // 'efgh'
```
results in b containing 'abcdefgh'

2.
```
CHARACTER*25 t1, t2, t3*15
t1 = 'Gone shopping'
t2 = 'blue jumpers are in there'
t3 = t1(1:2) // ' ' // t2(6:9) // t2(17:23) // ' lake'
```
What is stored in T3?

See Hierarchy of Operators (Appendix 2).

15.8 Comparing Character Expressions

Two character expressions may be compared in a relational expression. Both sides of the relational expression must be character expressions. 1977 Fortran, as well as introducing character variables, made it illegal to compare a character expression (for example something enclosed in quotes) with anything other than another character expression. Greater than and less than operators are allowed, and use the Ascii or Ebcdic collating sequence depending upon the machine used. So, for example, '9' is less than 'A', which is less than 'a'. If two character expressions of unequal length are compared, then the shorter one is considered to be extended by blanks. So when using an Ascii machine, the comparison `'ab' .LT. 'a'` is false because 'a' is extended by a blank, and blank comes before 'b' in the Ascii sequence.

15.9 Supplied Functions

1977 Fortran supplies functions to operate on character variables, array elements and expressions. The following abbreviations are used

I = integer
C = character
L = logical
param = parameter

Function Name	*Number of Parameters*	*Type of Parameters*	*Type of Result*	*Description*
ICHAR	1	C*1	I	Position of the param in the table of ASCII codes (Appendix 3), starting at position 0.
CHAR	1	I	C*1	Character in the ASCII code table indexed by the parameter.
LEN	1	C	I	Length of param, ie. number of characters.
INDEX	2	C	I	Starting position of param 2 within param 1 (= 0 if string not found).
LGE	2	C	L	.TRUE. if param 1 is lexically greater than or equal to param 2 , otherwise .FALSE.
LGT	2	C	L	Lexically greater than.
LLE	2	C	L	Lexically less than or equal.
LLT	2	C	L	Lexically less than.

15.10 Functions and Subroutines

A character function is a function whose type is character (the value returned by the function is a character string). The function is declared to be of type character, and the function name must also be declared in the calling program. The form is

```
CHARACTER*length FUNCTION name (dummy parameters)
```

where 'length' is an integer constant. The calling program must also declare the function type, with

```
CHARACTER*length  name
```

15.10.1 Passing Character Parameters

Passing characters can be a painful process, because you need to know the length of the character variable in order to declare it within the function or subroutine. Fortran has a mechanism for allowing an 'adjustable' character length for parameters, so that the length used will be the length of the actual parameter for each subprogram reference. This is done by declaring (in the subprogram) the dummy parameter as

```
CHARACTER*(*) list of dummy parameters
```

For example

```
SUBROUTINE x (line, word)
CHARACTER*(*) line, word(10)
```

declares `line` as a character variable and `word` as a character array.

15.11 Sample Program

The following program will read a list of names and addresses and print them out in a presentable form. The name on the input line occupies the first 36 columns and the address takes columns 37 to 80 inclusive. The end of the data is indicated by a blank name field.

```
c     Author:  L. Landau
c     Date:    October 1979
c     Mods:    January 1981 to update to 1977 Fortran
c
c     Input Description:
c          Each line contains names and addresses. The last line
c          contains blanks in the name field.
c
c          columns        meaning
c           1 - 36        name of person
c          37 - 80        address of person
c
c     Purpose:
c          To read in and list peoples names and addresses
c          having 50 per page, with page counts at the top
c
      CHARACTER*44 addr
      CHARACTER*36 name
c
      ipage = 1
      lincnt = 0
```

```
c
c     write out a heading
c
      WRITE(6,10) ipage
10    FORMAT('1Name',46x,'Address',20x,'Page: ',i4/
     $       1x,'----',46x,'-------'//)
c
c     read the next name and address
c
20    CONTINUE
      READ (5,30) name, addr
30    FORMAT(A36,A44)
c
c     test for end of data
c
      IF (name .EQ. ' ') STOP
c
c     check if a heading is required, then write out name
c     and address and then go back for more
c
      IF (mod(lincnt,50) .EQ. 0) THEN
         ipage = ipage + 1
         WRITE (6,10) ipage
      END IF
      WRITE (6,50) name, addr
      lincnt = lincnt + 1
50    FORMAT(1X,A,14X,A)
      GO TO 20
      END
```

As an exercise suggest how you could modify this program to write out the address immediately following the name, ignoring the trailing blanks in the name field. So the output would be

```
Nit Allen, 23 Waldorf Grade Hollyoak Drain
```

instead of

```
Nit Allen                                23 Waldorf Grade Hollyoak Drain
```

15.12 Exercises

Exercise 15A

Write a program that will read in a line of text, and then will write out

```
HAPPY BIRTHDAY text
```

Input is terminated by an end of file. Use the program to print out

```
HAPPY BIRTHDAY to you
HAPPY BIRTHDAY to you
HAPPY BIRTHDAY dear Erin
HAPPY BIRTHDAY to you
```

Exercise 15B

Write a subroutine that will return the first word in a line of text that is passed to it as a parameter. A word may be delimited by a blank, a comma or a full stop. You can assume that the line has a maximum of 80 characters. If no delimiter is on the line, then just return the value of the line itself. The answers present a more general version of this subroutine, and its calling program. When you have answered the question, look carefully at the presented answer.

Exercise 15C

Write a program to read in a value for `i`. If the value is between 1 and 12 (inclusive), print the name of the i'th month, abbreviated to 3 letters. This program can and should be done without the use of `GO TO` statements.

Exercise 15D

Given an integer variable `j` with `j` not less than 1 and not greater than 7, set up a program to print one of the words MONDAY, TUESDAY, etc., depending on the value of `j`.

Exercise 15E

A company manufactures `n` products where `n` is not greater than 50. Each product has a 5-character code and a 20-character description. Write the following main program and subroutine. The main program reads the product information into two one-dimensional arrays of length 50, with codes in the first array and the corresponding descriptions in the second array. It then should read in a product code and call the subroutine which searches the first array, finds the product, and prints out the code and description. The main program should be capable of reading any number of product codes.

Exercise 15F

Write a program to read an integer which represents the day in the year 1981. Print out the the corresponding date in the form

```
day of week     day of month     month     year
```

For example, if the input were

```
328
```

the output should be

```
Tuesday   24TH November  1981
```

Exercise 15G

Write a program that will read with the A field descriptor a line of data containing integers in free field format. The data may contain the ten decimal digits, plus and minus signs, and commas. The program is to convert the characters into the corresponding integer numbers, and print out the original data and the computed integer values.

Chapter 16

Additional Fortran

The topics here are considered to be of secondary importance to the new programmer, but they are very useful.

16.1 More Format Descriptors

16.1.1 E Format

A computer is able to represent very large, and very small real numbers. The F format descriptor is not convenient for reading and writing these numbers. The E format descriptor may be used with real variables to read or write numbers, so that they appear with an *exponent*. The form of this format descriptor is

```
Ew.d
```

The 'w' and the 'd' have the same meaning as for F format. If E format is used to output a number the form of output will be

```
.nnnnn+eee
```

The 'nnnnn' are the digits after the decimal place. Plus or minus 'eee' is the exponent for the decimal number, and is so arranged that the first 'n' of the 'nnnnn' is non-zero. There will *never* be any digits to the left of the decimal point. The number of places after the decimal point is specified by 'd'. Note that a space must be left for a possible minus sign to the left of the number, so there are a possible 6 extra places taken up other than the d decimal places and so 'w' must be at least six greater than 'd'. The statements

```
      first = 2.3756
      sec   = -677.32e-12
      third = 3444.55e18
      WRITE(6,101) first,sec,third
101   FORMAT(1X,E12.3,E20.6,E14.7)
```

will print the line

␣␣␣␣.238+001␣␣␣␣␣␣␣␣-.677320-009␣␣.3444550+022

On input, if the decimal point is in the data it overrides the 'd' specification of the E format descriptor. The exponent need not be three digits, and the number must be right-justified in the field (otherwise the blanks at the right end of the field are read as zeros, and included in the exponent).

16.1.2 L Format

Logical values (true and false) may be read or written with the L format, which has the form

```
Lw
```

where 'w' is an integer representing the width of the field. On input, blanks and/or a decimal point are allowed to precede a T or F, and the rest of the field is ignored. So with a format descriptor of L7, the following are all read as true,

```
␣.TRUE.
␣␣␣T
THAT'S
```

whilst the following are all read as false.

```
.FALSE.
F
␣␣␣␣FAT
```

On output, 'w'-1 blanks are written, followed by a T or F.

16.2 More Data Types

Two other data types exist in Fortran.

16.2.1 Double Precision

The data type `DOUBLE PRECISION` may be used if the size of a variable will exceed the maximum limit allowed in single precision or will be too small to be represented in single precision (i.e. underflow). More often it is the case that rather than the order of magnitude causing problems, it is the maximum number of significant figures (depleted by truncation and rounding errors) that motivates the move to double precision. The disadvantages of using `DOUBLE PRECISION` are

1. That each double precision variable requires twice as much memory as compared with that required by ordinary `REAL`s.
2. That the time taken for double precision arithmetic is appreciably longer than the time for the corresponding operation on `REAL`s.

Double Precision Constants These are very similar in form to the exponent form of real constants, except that there is a D instead of the E. The general form is

```
nnnn.nnnnD+eee
```

The n's are digits comprising the base value of the number, and 'eee' is the exponent part of the number. For example the number 1.23456 would be represented as 1.23456D0 as a double precision constant. Other examples are

```
21.34D-10
123456789.987654D-127
1.0D0
```

Double Precision Variables A double precision variable is formed in the normal way and is declared to be double precision by the type declaration

```
DOUBLE PRECISION variable-list
```

Arithmetic expressions are formed in double precision according to the same rules that apply to real expressions. The arithmetic operators (+,-,*,/,**) are the same. The combination of double precision with real and integer constants and variables in an arithmetic expression is explicitly permitted. In both cases, the result is always a double precision value. It is permissible to have an integer, real, or double precision variable on the left side of an arithmetic statement and an expression of some other type on the right side. All arithmetic is done according to the type of the expression on the right, and the result is converted according to the variable on the left. Acceptable uses of double precision quantities are

```
DOUBLE PRECISION d1,d2,d3,d4,d5,d6
d1 = d2*d3+(d4-8756.7865432D0)/d5
d1 = 4.0*d2-d3/1.1d0
d1 = r1+d1+r2
r1 = (d1*d2-d3*d4)/(d1*d5-d3*d6)
d1 = r1+2.0
d1 = (i1-8)*i2
i1 = r1+d1
d1 = d2**2
```

Double Precision Functions Generic functions, described in chapter 6, may have double precision parameters and/or return double precision results. Some of these functions are

`ABS(double)`	returns the absolute value of a double precision number
`MOD(dbl1,dbl2)`	remainder on dividing double precision numbers
`MAX(dbl1,dbl2,...)`	returns largest of double precision numbers
`MIN(dbl1,dbl2,...)`	returns smallest of double precision numbers
`DBLE(real or int.)`	returns double precision equivalent of real or integer number
`INT(double)`	truncates double precision numbers to integer
`REAL(double)`	returns most significant part of double precision number as a real number
`SIN(double)`	returns sine of double precision number expressed in radians
`COS(double)`	returns cosine of double precision number expressed in radians
`EXP(double)`	returns exponential of double precision number
`SQRT(double)`	returns square root of double precision number

Input/Output Input and output of double precision quantities is handled with the D field descriptor which is similar to the E field descriptor except that

1. the list variable associated with this field descriptor must be double precision,
2. there may be more digits, and
3. D is used for the exponent indicator rather than E.

16.2.2 Complex

This data type is used to represent a complex number (in mathematical terms, comprising a real and an imaginary part). The representation is as a pair of real numbers. **COMPLEX** constants are represented as a pair of real numbers separated by a comma and enclosed in brackets. complex variables must be declared in a type **COMPLEX** declaration statement. Input/output is accomplished by using two real field descriptors for each complex value written or read. For example consider the statements

```
      COMPLEX freud, fraser, quincy
      READ(5,100) freud
100   FORMAT(2F10.0)
      quincy = (1.2,-3.4679)
      fraser = quincy*freud + 2.5
```

Mixed mode (with the exception of exponentiation) is allowed between type **COMPLEX** and **REAL** or **DOUBLE PRECISION**, and the result will be **COMPLEX**. A complex variable or constant may only be assigned to a complex variable.

Complex Functions The generic functions for **COMPLEX** data types are

`REAL(complex)`	returns the REAL part of a COMPLEX number.
`AIMAG(complex)`	returns the imaginary part of a COMPLEX number.
`CMPLX(a,b)`	express two real numbers or two integers in complex form.
`CONJG(complex)`	obtain the complex conjugate of a complex number.
`INT(complex)`	returns the real part, truncated to an integer.
`ABS(ar,ai)`	returns the real result of `SQRT(ar**2 + ai**2)`, where `ar` is the real part and `ai` is the imaginary part of a complex number.
`SQRT, EXP, LOG, SIN or COS(complex)`	are as for other data types.

16.3 Data Statement

The **DATA** statement is used to assign initial values to variables. This assignment is done at the time of compilation and not at the time of execution of the compiled program. The DATA statement is not an executable statement. The form of the statement is

```
DATA data-list, data-list, ...
```

where a data-list is a list of the form

```
variable-list / value-list /
```

The variables in the variable-list are assigned the corresponding values of the constants in the value-list. There must be a one to one correspondence between the variables and the values, and generally the type of a variable and its value must be

the same. If the types do not match then the value is converted to the variable type if possible (e.g. integer to real), or if conversion is not possible, the initialization causes an error (e.g. integer to logical or character). The statement

```
DATA a,b,c /14.7,62.1,1.5E-20/
```

assigns the value 14.7 to a, 62.1 to b, and 1.5E–20 to c. The two statements

```
DATA a /67.87/, b /54.72/, c /5.0/
DATA a,b,c /67.87,54.72,5.0/
```

have the same effect, the choice being a personal preference. It is possible to repeat a value a number of times. For example, the statement

```
DATA r,s,t,u,v,w /6*21.7/
```

assigns the value 21.7 to all six variables. The repeat count must be an integer constant. As the DATA statement assigns initial values to variables, it is legal to redefine the values of these variables later in the program, but, having done so, it is not possible to 're-execute' the DATA statement to return the variables to their initial values. Initial values may be assigned to variable lists in any program unit. Initial values may not be specified for dummy parameters, and may not be specified for any variable more than once. A variable list on a DATA statement may include variables, arrays, array elements, substring names, and implied-do groups, separated by commas. The format of an implied-do group is

```
(variable-list, index = start, end, increment)
```

where start, end and increment are expressions containing only integer constants. The increment is optional and must be positive. Implied-do groups may be nested. Array subscripts must be integer expressions using only constants, parameter variables (see below) and implied-do index variables. Substring expressions must be integer constant expressions. If an entire array is initialized (with no implied-do), then the elements are initialized with the first subscript changing fastest, etc. (eg. down the columns in a 2D array). If the value list is too short, the last variables are not initialized and if it is too long, the last values are ignored. A DATA statement is placed after type and dimension declarations of variables which appear in the DATA statement. Example

```
REAL a(10), b(10)
CHARACTER*4 c(2)
DATA a / 10*0.0 /
DATA ( b(i), i = 1,5 ) / 5*1E5 /, (b(i), i = 6,10) / 5*2E10 /
DATA c(1)(1:2), c(2) /'ab', 'cdef'/
```

16.4 Implicit

The IMPLICIT statement assigns a data type to a name depending on the initial letter of the name. It has the form

```
IMPLICIT  type (letters), type (letters)
```

where 'type' may include a length (eg. CHARACTER*4) and the letters are single letters or letter ranges separated by commas. A letter range consists of two letters separated by a hyphen. For example, B–F means B,C,D,E,F. IMPLICIT overrides

the default association of particular letters with data types. The letters in brackets become associated with the specified type. Letters which are not included on an **IMPLICIT** statement retain their default types. Names affected are all variables, arrays, parameter constants, functions and statement functions within the program unit. The **IMPLICIT** statement is placed before type declarations and **DATA** statements. It may appear after a **PARAMETER** statement, in which case default types apply to the parameters. Example

```
IMPLICIT LOGICAL (L)
IMPLICIT CHARACTER*4 (C-E),  COMPLEX (F-H,X-Z)
```

After this, A,B,O–W still default to real numbers and I–K,M,N to integers. The usual method of converting a program to use double precision instead of real numbers is to use

```
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

16.5 Parameter

The **PARAMETER** statement allows constants in a program unit to be referenced by names, in order to make a program easier to alter. It has the form

```
PARAMETER ( n1 = e1, n2 = e2, ... )
```

where **n1, n2, ...** are variable names and **e1, e2, ...** are constant expressions. The **PARAMETER** statement is not executable, and is placed in the program before the parameter variables are referenced. If a parameter variable is not to have a default type, then its type must be declared (with **IMPLICIT** or an explicit type declaration) before the **PARAMETER** statement. The constant expression is evaluated at compile time, and may consist of constants, parameter variables defined on previous **PARAMETER** statements, or Fortran-supplied functions. A parameter variable cannot be redefined in the program. Each reference to a parameter variable in the program unit is replaced by its constant value. It is usually used in declaring array sizes in the main program, or to define the length of a character variable or array, by putting the parameter variable in brackets. It cannot be used as a constant in a **FORMAT** statement. Example

```
PARAMETER ( n = 100, l = 80 )
REAL array(n)
CHARACTER*(L)  line
DATA array / n * 0.0 /   len / l /
```

16.5.1 Example Using Double Precision, Data and Parameter

Read pairs of X and Y co-ordinates, one pair per line, in double precision. The first line of data is the number of points (i.e. X,Y co-ordinates) to be read in. Find the average Y value. Then for each point, print X, Y and

$$\frac{Y - average}{\pi \times X}$$

```
      PARAMETER (nmax = 30)
      DOUBLE PRECISION x(nmax), y(nmax), f, pi, ave
      DATA pi/3.14159265D0/, ave/0.0D0/
      READ(5,*) n
      IF (n .GT. nmax) THEN
         WRITE (6,*) 'max. no. of points allowed is', nmax
         STOP
      END IF
      READ (5,10) (x(i),y(i), i = 1,n)
10    FORMAT(2F15.9)                          or      FORMAT(2D15.9)
      DO 30 i = 1, n
         ave = ave + y(i)
30    CONTINUE
      ave = ave / n
      DO 50 i = 1, n
         f = (y(i) - ave) / (pi * x(i))
         WRITE (6,*) x(i), y(i), f
50    CONTINUE
      STOP
      END
```

16.6 External

EXTERNAL is a specification statement, and must precede all executable statements. It indicates that a name is a subroutine or function name, and not the name of a variable or Fortran-supplied function. It has the form

```
      EXTERNAL  names
```

where 'names' are subroutine or function names separated by commas. This statement is needed when the first reference to a subroutine or function does not have an explicit actual parameter list. For example, a subroutine or function which is not called from a program unit, but is passed to a subroutine or function as an actual parameter, is only referenced by its name, not with its parameter list, so the program unit needs to be able to distinguish it from a variable that has not been explicitly declared. Note that a function with no parameters need not have brackets on its **FUNCTION** statement, but when it is called it must have empty brackets (as shown in the example below). If a name is the same as a Fortran-supplied function, then the user-supplied function will be used instead of the Fortran-supplied function (eg. to write your own **SIN** routine). Example (see a full Fortran manual for a description of **COMMON**)

```
      EXTERNAL fun
      READ (5,*) a,b,c
      CALL sub(fun)
c     calls SUB with FUN as an actual parameter
      WRITE (6,*) a,b,c
      STOP
      END
C
      FUNCTION fun
      COMMON a,b,c
```

```
      fun = b**2 - 4*a*c
      RETURN
      END
C
      SUBROUTINE sub(f)
c     Function F is a dummy parameter.
      COMMON x,y,z
      z = SQRT( f())/ (2*x)
c     Call to function F needs empty parameter list
      x = SIN(z)
      y = COS(z)
      RETURN
      END
```

EXTERNAL tells the main program that fun is a function or subroutine, and not a real variable like x, y and z.

Appendix A

Notes on Doing Assignments

The assignments are designed to try to teach you some aspects of Fortran programming. Although it is important for your program to produce the correct answer, we are not interested in just this. We already know the answers, and know how to do the problems! It is how you arrive at the answers (i.e. the structure of the program, and to some extent, the method used in the program) that is important. The notes below are intended to let you know the kinds of things that will be looked for in marking your assignments. The comments below apply only to those questions that require the writing of a Fortran program.

1. The program must correctly solve the question asked. You should check your results. Some results are more easily checked than others. You should look very carefully at your answers and satisfy yourself that they are correct. If there is insufficient information printed out then you should change your program to print out sufficient information, before handing the assignment in.

2. The program should be written in such a way that it can be easily understood by you in a year's time, and also by anyone else (who has not read the assignment sheet) who understands Fortran, most importantly the person marking your assignment! This may be achieved by

   (a) Good program design

   (b) Choice of meaningful variable names

   (c) Well structured comments

3. At the beginning of the program you should use comments to explain

   (a) What the program is designed to do.

   (b) Who wrote it.

   (c) When it was written.

   (d) What general method the program employs.

   (e) How the data (if any) is designed, and how to use the program.

   (f) Any limitations the program has.

   In order to help you to remember to put in this information you should include the following titles within your comments, even if they do not seem to apply

```
C     Author:
C     Date:
C     Purpose:
C     Input Description:
```

All assignments handed in must have those titles at least, or they will be rejected. Of course you must also fill out the titles with appropriate information. Whenever helpful or necessary throughout the program you should use comments to explain

(a) What a variable is used to represent (unless obvious).

(b) Why a particular operation is being performed.

(c) Any special 'tricks' you use.

4. Headings on output. All the output from the program should be clearly headed so that even if you didn't have a program listing to accompany the output, it would be clear from your headings what the results are and how they are to be interpreted.

5. Program Generality. There are various degrees of program generality, and a balance needs to be found. Minimally, the program should be able to be run again, using different (although the same quantity of) data, *without any change to the program whatsoever!* Sometimes program changes may be necessary, in order that the program can run with an increased amount, or different type of data. In this case, the fewer changes required (generally) the more general the program. Remember that the suggested data for any program is only sample, and the program must be able to handle other data sets as well.

6. Elimination of unnecessary program statements. This is *not* to be taken to an extreme! A program with fewer statements may in fact be a much worse program both in the sense of understandability and efficency. An example of what will be looked for in this area is

> We have seen how it is possible to repeat a series of instructions in Fortran by means of a `DO` statement. Another means is to write out the statements explicitly the number of times that you wish them to be repeated. The latter is far worse.

If you have any queries as to what is expected in assignments, ask about it in tutorials. In spite of what you may think, the assignments are not meant to TEST you, but to help you LEARN.

Appendix B

Operator Hierarchy and Statement Order

B.1 Hierarchy of Operators

The following hierarchy is used to determine the order of evaluation of expressions

Rank	*Kind*	*Operation*
1	expressions in parentheses	all
2	functions	all
3	arithmetic	** (exponentiation)
4		*,/ (multiplication, division)
5		+,- (addition, subtraction)
6	character	// (concatenation)
7	logical	.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.
8		.NOT.
9		.AND.
10		.OR.

B.2 Ordering of Statements

1. **SUBROUTINE** or **FUNCTION** (for a subprogram)
2. **PARAMETER**
3. **IMPLICIT**
4. **DIMENSION** and type declarations
5. (**PARAMETER** may come after type declarations)
6. **EXTERNAL**
7. **DATA**

8. executable statements
9. **END**

Appendix C

Answers to Exercises

C.1 Chapter 2

Exercise 2A

1. `Z = (A+B)/(C+D)`
2. `Z = A + B/(C+D)`
3. `Z = (A+B)/C + D`
4. `Z = A + B/(C+D/E)`
5. `Z = N*(N-1)/2`
6. `Z = (A-B)*(C-D) / (E*(F+G))`
7. `Y = -2.314 + (5.67*Z - 3.29E-4)*Z**3 + 4.13*Z**7`

Exercise 2B

1. real
2. real
3. integer
4. neither
5. real
6. real
7. neither
8. neither
9. neither
10. neither
11. real

Exercise 2C

1. real
2. illegal
3. real
4. illegal
5. illegal
6. integer
7. illegal
8. real
9. illegal
10. illegal
11. real
12. real

Exercise 2D

1. 3B is an illegal variable name
2. There is a mismatch of open and close brackets
3. 3.14159 is an illegal variable name
4. Adjacent operators ** and –
5. Only 6 characters are allowed for variable names.
6. X+Y is an illegal variable name.
7. Missing operator between the two bracketted expressions

Exercise 2E

1. Z becomes –1.0666667 (approximately)
2. Z becomes 4.0 (note that this is NOT 4, but 4.0)
3. Z becomes 1.0
4. Z becomes 6.0
5. Z becomes 1.0
6. K becomes 3 (note that this is NOT 3.0, but 3)
7. K becomes 2
8. K becomes 1
9. K becomes 6
10. K becomes -1
11. K becomes 0

C.2 Chapter 3

Exercise 3A

```
C
C   AUTHOR:   LESLIE LANDAU
C   DATE:     OCTOBER 1977
C   INPUT DESCRIPTION:  FREE FORMAT, 2 REALS 2 INTEGERS
C   PURPOSE:
C     TO READ IN 4 NUMBERS AND TO WRITE THEM OUT IN REVERSE
C
      READ(5,*)VAL1,VAL2,NUM3,NUM4
      WRITE(6,*)NUM4,NUM3,VAL2,VAL1
      STOP
      END
```

Exercise 3B

```
C  AUTHOR:    LESLIE LANDAU
C  DATE  :    OCTOBER 1977
C  INPUT :    TWO INTEGERS IN FREE FORMAT
C  PURPOSE:
C           TO ADD, MULTIPLY, DIVIDE THE TWO INTEGERS
C           AND TO RAISE THE FIRST TO THE POWER OF
C           THE SECOND
C
      READ(5,*)I,J
      WRITE(6,*)' ORIGINAL INPUT IS ',I,J
      K = I+J
      L = I*J
      M = I/J
      N = I**J
C
C  WRITE OUT RESULTS
C
      WRITE(6,*)' ADDITION = ',K,' MULT = ',L,' DIV = ',M,
     $          ' POWER = ',N
      STOP
      END
```

Exercise 3C

```
C  AUTHOR:      L. LANDAU
C  DATE  :      OCT  1977
C  INPUT :      A POSITIVE REAL NUMBER IN FREE FORMAT
C  PURPOSE:
C           TO CALCULATE THE INTEGRAL AND FRACTIONAL PARTS
C           OF THE REAL NUMBER READ IN
C
      READ(5,*)VALUE
      INT = VALUE
      FRACT = VALUE - INT
```

```
      WRITE(6,*)' THE INTEGRAL PART OF ',VALUE,' IS ',INT,
     $          ' AND THE FRACTIONAL PART IS ',FRACT
      STOP
      END
```

Exercise 3D

```
C  AUTHOR :      L. LANDAU
C  DATE   :      OCT 77
C  INPUT  :      TWO REALS IN FREE FORMAT
C  PURPOSE:      TO CALCULATE THE DIFFERENCE BETWEEN
C                THE SQUARES ( FIRSTSQ - SECONDSQ)
C
      READ(5,*)VAL1,VAL2
      SQ1 = VAL1*VAL1
      SQ2 = VAL2**2
      DIFF= SQ1 - SQ2
      WRITE(6,*)VAL1,' SQUARED IS ',SQ1
      WRITE(6,*)VAL2,' SQUARED IS ',SQ2
      WRITE(6,*)' DIFFERENCE BETWEEN SQUARES = ',DIFF
      STOP
      END
```

C.3 Chapter 4

Exercise 4A

1. Valid
2. Valid
3. Invalid (Illegal adjacent operators)
4. Valid
5. Invalid (Missing arithmetic expression)
6. Valid
7. Invalid (Illegal operator)
8. Invalid (Illegal operator)

Exercise 4B

```
C
C  AUTHOR:  LES LANDAU
C  DATE:    9 SEP 79
C  INPUT:   ONE LINE CONTAINING TWO INTEGERS IN FREE FORMAT
C  PURPOSE: TO DETERMINE WHICH INTEGER IS THE LARGER, AND WRITE
C           THAT ONE OUT FIRST
C
      READ(5,*) INT1,INT2
```

```
      IF (INT1 .GT. INT2) THEN
          WRITE(6,*)' NUMBERS ARE ',INT1,INT2
      ELSE
          WRITE(6,*)' NUMBERS ARE ',INT2,INT1
      END IF
      STOP
      END
```

Exercise 4C

```
      READ (5,*) MAXIN
      IF (MAXIN .EQ. 10) THEN
          FORGET = 0.0
      ELSE IF (MAXIN .EQ. 16) THEN
          WRITE (6,*) 'Gotit'
          STOP
      ELSE IF (MAXIN .EQ. 19) THEN
          FORGET = 0.5
          WRITE (6,*) ' Found one'
      END IF
```

Exercise 4D

```
    IF (A.GT.-0.00001 .AND. A.LT.0.00001) A = 0.0
```

Exercise 4E

```
      IF (IND .EQ. 16) THEN
          K = 6
          I = 9
      ELSE IF (L .GT. J+4) THEN
          X = 19.6
          READ (5,*) Y
          L = 0
      ELSE IF (MAIN .LT. I) THEN
          COST = 19.0
          TRY  = 14.2
      ELSE
          PAY = 0.0
      END IF
```

Exercise 4F

```
C     Author : K. Handel
C     Date   : 30-8-84
C     Purpose : To add, subtract, multiply or divide two numbers A and
B
C               which are read from input.
C     Input   : On line 1, enter an integer code, in free format,
C               representing the type of operation.
C               On another line, enter the numbers to operate on.
```

```
C            The codes are :  1 for A + B
C                             2 for A - B
C                             3 for A * B
C                             4 for A / B
      WRITE (6,*) 'Enter one of the following codes :'
      WRITE (6,*) '    1 to add two numbers'
      WRITE (6,*) '    2 to subtract the 2nd number from the 1st'
      WRITE (6,*) '    3 to multiply two numbers'
      WRITE (6,*) '    4 to divide the 1st number by the 2nd'
      WRITE (6,*)
      WRITE (6,*) 'Which code do you want?'
      READ  (5,*) ICODE
      WRITE (6,*) 'Enter the 2 numbers, separated by a comma or blank:'
      READ  (5,*) A, B
      IF (ICODE .EQ. 1) THEN
         ANS = A + B
         WRITE (6,*) 'Sum of ', A, ' and ', B, ' is ', ANS
      ELSE IF (ICODE .EQ. 2) THEN
         ANS = A - B
         WRITE (6,*) 'Difference between ', A, ' and ', B, ' is ', ANS

      ELSE IF (ICODE .EQ. 3) THEN
        ANS = A * B
        WRITE (6,*) 'Product of ', A, ' and ', B, ' is ', ANS

      ELSE IF (ICODE .EQ. 4) THEN
        IF (B .EQ. 0) THEN
           WRITE (6,*) 'Can''t divide by zero!'
           STOP
        ELSE
          ANS = A / B
          WRITE (6,*) 'Quotient of ', A, ' on ', B, ' is ', ANS
        END IF

      ELSE
        WRITE (6,*) ICODE, ' is an illegal code!'
      END IF

      STOP
      END
```

C.4 Chapter 5

Exercise 5A

1. FALSE
2. TRUE
3. FALSE
4. FALSE

5. TRUE
6. FALSE
7. FALSE
8. TRUE
9. FALSE
10. FALSE

Exercise 5B

1. correct
2. incorrect (statement label is a variable)
3. correct
4. incorrect (DO-variable cannot be a number)
5. correct
6. correct (real number is truncated to an integer)
7. correct
8. correct
9. correct (1, N4A2 and 2 are converted to real numbers)
10. correct
11. incorrect (missing statement label)

Exercise 5C

1. 14
2. 0, 4, 8, 12, 17, 21, 25, 29

Exercise 5D

1. 12
2. 12
3. 9
4. 15
5. 4

Exercise 5E

44 because J is incremented to 44, then tested against 38, and control is passed to the WRITE statement.

Exercise 5F

```
C
C   AUTHOR:     LES LANDAU
C   DATE:       9 SEP 79
C   INPUT:      THERE IS NO INPUT
C   PURPOSE:    TO ADD UP ALL THE EVEN INTEGERS BETWEEN 98 AND 224
C               INCLUSIVELY, AND TO WRITE OUT THE TOTAL.
C
      ITOTAL = 0
      DO 5 NUM = 98,224,2
         ITOTAL = ITOTAL + NUM
  5   CONTINUE
C
C   WRITE OUT THE RESULTANT TOTAL
C
      WRITE(6,*) ' SUM OF EVEN INTEGERS FROM 98 TO 224 = ', ITOTAL
      STOP
      END
```

Exercise 5G

```
C
C AUTHOR:    LES LANDAU
C DATE:      9 SEP 79
C INPUT:     FIRST LINE:
C                THIS CONTAINS AN INTEGER IN FREE FORMAT WHICH
C                INDICATES THE NUMBER OF LINES TO FOLLOW
C            SUBSEQUENT LINES:
C                CONTAIN A REAL NUMBER (FREE FORMAT)
C
C PURPOSE:   TO DETERMINE THE LARGEST AND SMALLEST NUMBER
C
C RESTRICTION:  THERE MUST BE AT LEAST ONE NUMBER
C
C
C    READ IN HOW MANY NUMBERS THERE ARE
C
      READ(5,*) NUM
      IF (NUM .LT. 1) THEN
          WRITE(6,*)' YOU NEED AT LEAST ONE NUMBER!!'
          STOP
      END IF
C
C    READ IN THE FIRST NUMBER AND SAY ITS BOTH THE BIGGEST AND
C    THE SMALLEST.
C
      READ(5,*) VAL
      BIG = VAL
      SMALL = VAL
C
C   NOW PROCESS THE REMAINING NUM-1 NUMBERS
```

```
C
      DO 5 L = 2,NUM
         READ(5,*) VAL
         IF(VAL.GT.BIG) BIG = VAL
         IF(VAL.LT.SMALL) SMALL = VAL
  5   CONTINUE
C
C     WRITE OUT RESULTS
C
      WRITE(6,*) 'THE LARGEST  NUMBER WAS ',BIG
      WRITE(6,*) 'THE SMALLEST NUMBER WAS ',SMALL
      STOP
      END
```

Exercise 5H

```
C     Purpose : calculate depth of a well
C     Input   : none
C     Author  : K. Handel  30-8-84
      G = 9.807
      S = 331.6
      S2G = S**2/G
      DO 25 T = 1,3
        D = S2G + S*T - S2G*(1+2*G*T/S)**0.5
        WRITE(6,*) 'Depth at time ',T,'is ',D
 25   CONTINUE
      STOP
      END
```

C.5 Chapter 6

Exercise 6A

1. `ANS = SQRT(B*B - 4.0*A*C)`

2. `ANS = SIN(2.4)`

3. `IANS= MAX(J,LARG)`

4. `ANS = MAX(A,BIG)`

5. `ANS = ABS(ECC)`

6.
```
      IANS= ABS(I)
      JANS= ABS(M)
      ANS = ABS(A)
```

7. `ANS = REAL(KKK)`

8. `ANS = SQRT(REAL(INT))`

9. `ANS = MAX(REAL(I),X)`

Exercise 6B

```
C  AUTHOR  :      L.LANDAU
C  DATE    :      7TH OCTOBER 1977
C  INPUT   :      ONE INTEGER IN FREE FORMAT
C  PURPOSE :      TO CALCULATE THE FACTORIAL OF THE INTEGER
C                 READ IN
C  RESTRICTIONS:
C                 THE UNIVAC COMPUTER CAN ONLY REPRESENT
C                 INTEGERS UP TO 2**35 -1 AND SO THE
C                 MAXIMUM FACTORIAL IS 12!
C
      READ(5,*)NUM
C
C  STOP THE PROGRAM IF NUM IS TOO BIG
C
      IF (NUM .GT. 12) THEN
          WRITE(6,*)NUM,' IS TOO BIG, 12 IS MAX'
          STOP
      END IF
      IFACT = 1
      DO 5 MULT = 2, NUM
         IFACT = IFACT*MULT
 5    CONTINUE
C
C  OUTPUT  RESULTS
C
      WRITE(6,*)' THE FACTORIAL OF ',NUM,' IS ',IFACT
      STOP
      END
```

Exercise 6C

1. `M = MOD(IJ,K)`

2. `SMALL = MIN(A,B)`

3. `WARM  = REAL(ION)/REAL(KAN)`

Exercise 6D

```
C   AUTHOR : K. HANDEL
C   DATE   : 28 JUNE 81
C   INPUT  : (1)  AN INTEGER IN FREE FORMAT INDICATING THE NUMBER
C                 OF VALUES TO FOLLOW
C            (2)  THE VALUES, REAL NUMBERS, ONE PER LINE
C
C   PURPOSE : FIND THE AVERAGE OF ANY NUMBER OF NUMBERS READ IN
C             FROM INPUT
C   RESTRICTION : THERE MUST BE AT LEAST ONE NUMBER
C
```

```
      TOTAL = 0

      READ (5,*) NUM
      DO 100 I = 1,NUM
         READ (5,*) VAL
         TOTAL = TOTAL + VAL
  100 CONTINUE

      AVER = TOTAL/NUM
      WRITE (6,*) 'AVERAGE IS  ',AVER
      STOP
      END
```

Exercise 6E

```
C   Purpose : Find the average and standard deviation of numbers
C             which are read from input.
C   Input   : (1) An integer in free format indicating the number
C                 of values to follow.
C             (2) Values to be averaged, one real number per line
C                 in free format. End of data is end-of-file marker.
C   Author  : K. Handel   29-8-84
C   TOTAL is the sum of the values
C   SS    is the sum of the squares of the values
C   NUM   is the number of values
      TOTAL = 0
      SS    = 0

      READ (5,*) NUM
      DO 10 I = 1,NUM
        READ (5,*) VAL
        TOTAL = TOTAL + VAL
        SS    = SS + VAL**2
        NUM   = NUM + 1
10    CONTINUE

      AVER = TOTAL/NUM
      SD   = SQRT( SS/NUM - AVER**2)
      WRITE (6,*) 'Average is', AVER
      WRITE (6,*) 'Standard deviation is', SD
      STOP
      END
```

C.6 Chapter 7

Exercise 7A

```
C
C  AUTHOR:   L. LANDAU
C  DATE:     9 SEP 79
C  INPUT:    THREE REAL NUMBERS AND ONE INTEGER
```

```
C          IN FREE FORMAT. INPUT FOR VARIABLES
C          A, B, P, K  (RESP)
C
C  PURPOSE:  TO DO ONE OF THREE CALCULATIONS
C           DEPENDING ON K BEING -VE,0,+VE
C
      READ(5,*)A, B, P, K
      IF(K.LT.0) Y = SQRT(A**2 + SIN(P)**2)
      IF(K.GT.0) Y = SQRT(B**2 - SIN(P)**2)
      IF(K.EQ.0) Y = SQRT(A**2 + B**2)
C
C  WRITE OUT RESULTS
C
      WRITE(6,*) 'PARAMETERS READ IN : A = ',A,
     $              'B = ',B,'P = ',P,'K = ',K

      WRITE(6,*) 'CALCULATED VALUE, Y = ',Y
      STOP
      END
```

Exercise 7B

```
C  AUTHOR:  K.HANDEL
C  DATE:    5 AUGUST 1981
C  INPUT:   NONE
C
C  PURPOSE: CALCULATE E TO THE POWER OF X
C           FOR X AN INTEGER RANGING FROM 1 TO 100,
C           USING THE EXP FUNCTION.
C  RESTRICTION: X WILL BECOME TOO LARGE FOR EXP
C               AT SOME STAGE. X COULD BE MADE
C               INTO DOUBLE PRECISION TO EXTEND
C               THE RANGE.

      INTEGER X
      DO 10 X = 1,100
         WRITE (6,*) X,EXP(X)
   10 CONTINUE
      STOP
      END
```

Exercise 7C

Algorithm

1. Initialize MAXAGE and MINAGE to impossible ages (or to the first person's age). Assume there is at least one person.
2. Read id and age for a person. If end of data, go to 5.
3. If age is a new minimum, update MINAGE and corresponding id.
4. If age is a new maximum, update MAXAGE and corresponding id.

5. Print MINAGE, MAXAGE and corresponding id's.

Program

```
C  Author  : K. Handel  Aug 1984
C  Input   : One line per person. Each line has:
C            identification number and age (2 integers in free
C            format). End of data is an end-of-file marker.
C  Purpose : Find maximum and minimum ages and corresponding id's
C  Initialize to impossible ages
      MAXAGE = -1
      MINAGE = 999
 15   READ(5,*,END=25) ID, IAGE
      IF (IAGE.LE.MINAGE) THEN
         MINAGE = IAGE
         MINID = ID
      END IF
      IF (IAGE.GT.MAXAGE) THEN          (cannot use ELSE IF in case
         MAXAGE = IAGE                   the first person is the
         MAXID = ID                      oldest)
      END IF
      GO TO 15
 25   WRITE(6,*) 'Youngest person is ',MINID,' with age ',MINAGE
      WRITE(6,*) 'Oldest person is ',MAXID,' with age ',MAXAGE
      STOP
      END
```

Alternatively:

```
C  Initialize to age of first person
      READ (5,*)  ID, IAGE
      MAXAGE = IAGE
      MINAGE = IAGE
      MAXID  = ID
      MINID  = ID
 15   READ (5,*,END=25) ID, IAGE
      IF (IAGE.LT.MINAGE) THEN
         MINAGE = IAGE
         MINID = ID
      ELSE IF (IAGE.GT.MAXAGE) THEN
         MAXAGE = IAGE
         MAXID = ID
      END IF
      GO TO 15
 25   WRITE(6,*) MINID, MINAGE, MAXID, MAXAGE
      STOP
      END
```

Exercise 7D

Algorithm

1. Read number of people (N).

2. Initialize number of people over 30 (NOLD).
3. For each person,
   (a) read ID, AGE, SEX (sex is integer).
   (b) write ID, AGE, SEX (in characters).
   (c) if AGE > 30 then increment NOLD.
4. Write NOLD.

Program

```
C  Author  : K. Handel  August 1984
C  Purpose : For people in a survey, print the identification
C           number, age and sex of each person.
C           Also print the number of people over 30.
C  Input   : First line contains the number of people
C           (an integer in free format).
C           Then, one line per person. Each line has
C           ident., age and sex (3 integers in free format).
      INTEGER AGE, SEX
      NOLD = 0
      READ(5,*) N
      DO 100 I = 1, N
         READ (5,*) ID, AGE, SEX
         IF (SEX.EQ.0) THEN
            WRITE(6,*) ID, AGE, ' Male'
         ELSE
            WRITE (6,*) ID, AGE, ' Female'
         END IF
      IF (AGE .GT. 30) NOLD = NOLD + 1
  100 CONTINUE
      WRITE (6,*) 'Number of people over 30 = ', NOLD
      STOP
      END
```

Exercise 7E

```
C  AUTHOR:   L. LANDAU       9 SEP 79
C           modified by K. Handel 30-8-84
C  INPUT:    X, A REAL NUMBER IN FREE FORMAT
C  PURPOSE:  TO EVALUATE THE SUM OF THE SERIES :
C                        N
C                    COS (X)
C                    -------                FOR N = 0 TO 20
C                      N!

      READ (5,*) X
      SUM  = 1.0
      CX   = COS(X)
      TERM = 1.0
C
```

```
C  SUM THE SERIES
C
      DO 5 NTERMS = 1, 20
         TERM = TERM * CX / NTERMS
         SUM  = SUM + TERM
   5  CONTINUE
C
C  WRITE OUT RESULTS
C
      WRITE(6,*) 'THE VALUE OF F USING X = ',X,' IS ',SUM
      STOP
      END
```

The factorial is not calculated in a separate variable because it overflows at 13!

C.7 Chapter 8

Exercise 8A

1. legal
2. illegal
3. illegal
4. illegal
5. legal
6. legal
7. illegal

Exercise 8B

1. yes, 8A(a)
2. no
3. yes, 8A(e)
4. yes, 8A(f)

Exercise 8C

```
C  program to call subroutine INVERS with specific values
C  Author : K. Handel   21-8-84
C  Input  : none

      x = 5
      call invers(x,y)
      write (6,*) 'Inverse of', x, 'is', y
      x = 0
      call invers(x,y)
```

```
      write (6,*) 'Inverse of', x, 'is', y
      call invers(0.5,y)
      write (6,*) 'Inverse of 0.5 is', y
      stop
      end

      subroutine invers(x,y)
c  input parameter : x
c  output parameter: y, the inverse of x

      if ( x .eq. 0 ) then
         y = 0
      else
         y = 1/x
      end if
      return
      end
```

Exercise 8D

```
      SUBROUTINE TRIG (ANGLE,IDEGS,MINS,ISECS)
C  AUTHOR:       K. HANDEL
C  DATE:         5 AUG 1981
C  INPUT:        None
C  PURPOSE:      Convert an angle in degrees (real number)
C                into degrees, minutes and seconds (integers).
C                Seconds are rounded to the nearest integer.
C  PARAMETERS:
C       INPUT:   ANGLE - angle in degrees (REAL)
C      OUTPUT:   IDEGS,MIN,ISECS - integer number of degrees,
C                minutes and seconds.
      IDEGS = ANGLE
C  After subtracting the whole degrees, convert the remainder
C  to minutes (REAL).
      REM = (ANGLE - IDEGS) * 60
      MINS = REM
C  After subtracting the whole minutes, convert the remainder
C  to seconds.  0.5 is added before truncating the seconds to
C  an integer, so the effect is rounding to the nearest integer.
      ISECS = (REM - MINS) * 60 + 0.5
C     Round to whole minutes and/or degrees if necessary
      IF (ISECS .EQ. 60) THEN
         ISECS = 0
         MINS = MINS + 1
         IF (MINS .EQ. 60) THEN
            MINS = 0
            IDEGS = IDEGS + 1
         END IF
      END IF
      RETURN
      END
```

Exercise 8E

```
      SUBROUTINE FACT(N,NFACT)
C  N! overflows for N > 12.
      IF (N .GT. 12) THEN
         WRITE (6,*) N, ' is too big. 12 is max.'
         NFACT = 0
         RETURN
      END IF
      NFACT = 1
      DO 25 I=2,N
         NFACT = NFACT*I
 25   CONTINUE
      RETURN
      END
```

Exercise 8F

```
C                n              n          n!
C    Calculates  C    where     C   =  --------
C                 r              r      r!(n-r)!
C    for non-negative integers n and r.
C    Input : n and r in free format.
C    K. Handel   28-8-84
      INTEGER R, RFACT
      READ(5,*) N, R
      IF (N.LT.0 .OR. R.LT.0 .OR. N.LT.R) THEN
         WRITE(6,*) 'I can''t do that'
         STOP
      END IF
      CALL FACT(N, NFACT)
      CALL FACT(R, RFACT)
      CALL FACT(N-R, NRFACT)
      NCR = NFACT/RFACT/NRFACT
      WRITE(6,*)'There are ',NCR,' ways of choosing ',R,
     $ 'objects from ', N, 'objects'
      STOP
      END
C
      SUBROUTINE FACT (N,NFACT)
C  N! overflows for N > 13.
      IF (N .GT. 13) THEN
         WRITE (6,*) N, ' is too big. 13 is max.'
         NFACT = 0
         RETURN
      END IF
      NFACT = 1
      DO 25 I = 2, N
         NFACT = NFACT*I
 25   CONTINUE
      RETURN
      END
```

C.8 Chapter 9

Exercise 9A

1. There is no field descriptor in the FORMAT corresponding to the variable K. This may be fixed by putting in a field descriptor for K or by eliminating K from the WRITE statement.
2. (a) GATHER requires an F field descriptor
   (b) MOSS requires an I field descriptor
   (c) The field descriptor F3.5 doesn't make sense.
3. (a) There is no terminal statement for the DO
   (b) There is no point in setting I to 17 outside the DO as we read in a value for it within the DO.
   (c) We cannot read in a value for K within the DO as K is the control variable (and we are no allowed to change it).
   (d) As F is not used within the loop, its value will be overwritten each time around the loop.
   (e) The writing out of I requires an I field descriptor.
   (f) The construction .=. is not allowed and should be .EQ.

Exercise 9B

There will be 4 lines printed (including blank lines).

Exercise 9C

```
    0   10 -58790062*****
.00  16.77-586.21   5.48 131.10
```

Exercise 9D

1. `676.7`
2. `6381`
3. `␣␣␣␣␣132.6`
4. `****` (need F5.2 to get 12.16)
5. `␣␣␣␣␣99999`

Exercise 9E

```
C
C AUTHOR:   LES LANDAU
C DATE:     9 SEP 79
C INPUT:    TWO REAL NUMBERS IN FREE FORMAT
C           THE FIRST IS THE LENGTH OF CUBE A
C           THE SECOND IS THE SMALLEST SIDE OF BLOCK B
C
```

```
C PURPOSE:  TO CALCULATE THE SURFACE AREAS OF CUBE A AND
C           BLOCK B, AND ALSO TO FIND THE LARGER ONE AND THE
C           DIFFERENCE IN AREA
C METHOD OF CALCULATION:
C           THE AREA OF A CUBE IS 6 TIMES THE SQUARE OF THE WIDTH
C           THE AREA OF THE BLOCK B IS GIVEN BY:
C           2*2K*3K + 2*2K*K + 2*3K*K   (i.e. 22*K*K)
C           WHERE K IS THE LENGTH OF THE SHORTEST SIDE
C
      READ(5,*) H, BK
      AREAA = 6 * H**2
      AREAB = 22 * K**2
      DIFF  = ABS(AREAA - AREAB)
      BIG   = MAX(AREAA,AREAB)
C
C  NOW WRITE OUT THE RESULTS
C
      WRITE(6,100)
100   FORMAT('1','LES LANDAU',///
     $       52X,'COMPARISON OF SURFACE AREAS'///)
      WRITE(6,101) H, BK, BKH, BKL, AREAA, AREAB
101   FORMAT(1X,'SIDE OF CUBE A:',5X,F10.2,/
     $       1X,'WIDTH OF BLOCK B:',3X,F10.2/
     $       1X,'HEIGHT OF BLOCK B:',2X,F10.2/
     $       1X,'LENGTH OF BLOCK B:',2X,F10.2//
     $       1X,'SURFACE AREA OF A:',2X,F10.2/
     $       1X,'SURFACE AREA OF B:',2X,F10.2/)
      WRITE(6,102) DIFF,BIG
102   FORMAT(1X,'DIFFERENCE IN SURFACE AREA: ',3X,F10.2//
     +       1X,'LARGER SURFACE AREA: ',3X,F10.2)
      STOP
      END
```

Exercise 9F

```
C
C  AUTHOR:  L: LANDAU
C  DATE:    9 SEP 79
C  INPUT:   THERE IS NO INPUT
C  PURPOSE: TO PRINT OUT THE SINE, COSINE
C           TANGENT, SECANT, COSECANT, COTANGENT
C           OF EVERY INTEGRAL ANGLE FROM
C           1 TO 89 DEGREES

      WRITE(6,10)
 10   FORMAT('1','ANGLE',4X,'SINE',6X,'COSINE',
     $       6X,'TANGENT',7X,'SECANT',7X,'COSECANT',
     $       6X,'COTANGENT')

C   CALCULATE THE VALUE OF PI FROM ATAN FUNCTION
```

```
      PI = 4.0 * ATAN(1.0)
      DEGRAD = PI/180.0
      DO 15 IDEG = 1,89

C  FIND THE TRIG FUNCTIONS

        ANGLE = IDEG * DEGRAD
        S     = SIN(ANGLE)
        CS    = COS(ANGLE)
        T     = TAN(ANGLE)
        SEC = 1.0/CS
        COSC= 1.0/S
        COT = 1.0/T

C  WRITE  OUT  RESULTS

        WRITE(6,20)IDEG,S,CS,T,SEC,COSC,COT
   15 CONTINUE
   20 FORMAT(1X,3X,I2,3X,F7.4,4X,F7.4,4(4X,F10.4))
      STOP
      END
```

C.9 Chapter 10

Exercise 10A

1. A type F field descriptor is required for reading in a value for A
2. There is no such relational operator as .GRT. A field descriptor of I5.1 does not make sense. The format for writing out J should contain an I field descriptor.
3. Finishing a DO statement on a FORMAT statement is not a good practice. Cannot divide by zero (J/K is trying to do this)

Exercise 10B

1. 3
2. 2

Exercise 10C

1.
```
     I    WUNDA    WOT    IT      WILL     BE
   123  456.127    98.0    12      45.6   78.0
```

2.
```
     I    WUNDA    WOT    IT      WILL     BE
 12345   6.1279     8.0 12004     567.8    0.1
```

3.
```
     I    WUNDA    WOT    IT      WILL     BE
     1     23.4    56.1   279 8.00012004  5.67
```

Chapter 11

Exercise 11A

1. The array J is dimensioned to size 20 and yet the DO loop will index up to 100
2. The left hand side of the third line should have a subscript, as the array ARRAY must always appear with a subscript.
3. The variable I is being used as both an array AND as a simple variable. This is not allowed. ARRAY(I-1) when I is 1 will be referencing ARRAY(0) which is illegal.

Exercise 11B

```
C  AUTHOR:   K. HANDEL
C  DATE:     5 AUGUST 1981
C  INPUT:    Rainfall data for 8 localities, one line per locality.
C           12 real numbers per line, in free format,
C           representing the rainfall for each of the 12 months
C           for a locality.
C  PURPOSE:  calculate average rainfall for each of 8 localities.

      REAL RAIN(12)
      DO 40 LOCAL = 1,8
         READ (5,*) RAIN
         AVE = 0.0
         DO 20 MONTH = 1,12
            AVE = AVE + RAIN(MONTH)
20       CONTINUE
         WRITE (6,*) 'Average rainfall for locality',LOCAL,
     $               ' is', AVE/12.0
40    CONTINUE
      STOP
      END
```

Exercise 11C

1. correct
2. correct
3. incorrect
4. correct

Exercise 11D

1. X appears in a DIMENSION statement and so should have a subscript in the second line
2. no error
3. Cannot have a REAL variable as a subscript.

4. A variable may not appear twice on a DIMENSION statement. The maximum subscript for T is 6

Exercise 11E

```
      DIMENSION B(100)
      DO 5 LOC = 1,100
         B(LOC) = 0.0
5     CONTINUE
```

Exercise 11F

```
C
C  AUTHOR:  L  LANDAU
C  DATE  :  SEPT 1979
C  INPUT DESCRIPTION:
C   FIRST CARD:
C     CONTAINS AN INTEGER (IN FREE FORMAT)
C     WHICH INDICATES THE NUMBER OF CARDS
C     TO FOLLOW.
C
C   FOLLOWING CARDS:
C      COLUMNS      TYPE       MEANING
C       1-2         INTEGER    IDENTIFICATION NUMBER
C       3-4         INTEGER    NUMBER OF EGGS LAID
C                              THIS MONTH
C       5-8         INTEGER    FEED CONSUMED (GRAMS)
C       9-12        INTEGER    WEIGHT OF THE BIRD (GRAMS)
C
C  PURPOSE:
C  --------
C       READ IN HEN DATA AND PRINT OUT FOR EACH HEN
C       (A) IDENT
C       (B) EGGS LAID AND DIFFERENCE FROM AVERAGE
C       (C) FEED EATEN,   DIFFERENCE FROM AVERAGE
C       (D) BIRD WEIGHT,  DIFFERENCE FROM AVERAGE
C
C       THERE IS A LIMIT OF 50 HENS

      DIMENSION IDENT(50),LAYD(50),IFEED(50)
      DIMENSION IWEIGH(50)
C
C  READ IN NUMBER TO PROCESS AND ENSURE < 50
C
      READ(5,*)NUM
      IF(NUM.GT.50) THEN
         WRITE(6,*)' TOO MANY HENS, MAX IS 50. RECOMPILE',
     $             ' PROGRAM WITH LARGER ARRAYS'
         STOP
      END IF
      ITOTEG = 0
```

```
      ITOTFD = 0
      ITOTWT = 0
C
C  READ IN HEN DATA AND COMPUTE TOTALS
C
      DO 5 IHEN = 1,NUM
         READ (5,10) IDENT(IHEN),LAYD(IHEN),IFEED(IHEN),
     $                IWEIGH(IHEN)
         ITOTEG = ITOTEG + LAYD(IHEN)
         ITOTFD = ITOTFD + IFEED(IHEN)
         ITOTWT = ITOTWT + IWEIGH(IHEN)
  5   CONTINUE
 10   FORMAT(I2,I2,I4,I4)

C  WRITE OUT RESULTS
C
      WRITE(6,15)
 15   FORMAT('1','HEN ANALYSIS'/
     $       1X,11X,'EGGS    DIFFERENCE   FEED    ',
     $       'DIFFERENCE    BIRD    DIFFERENCE'/
     $       1X,' IDENT',5X,'LAID',4X,'FROM  AVE',
     $       4X,'EATEN   FROM  AVE',5X,'WEIGHT',
     $       3X,'FROM  AVE')
      AVEEG  = REAL(ITOTEG)/NUM
      AVEFD  = REAL(ITOTFD)/NUM
      AVEWT  = REAL(ITOTWT)/NUM
      DO 25 IHEN = 1,NUM
         DIFEG = LAYD(IHEN) - AVEEG
         DIFFD = IFEED(IHEN)- AVEFD
         DIFWT = IWEIGH(IHEN)-AVEWT
         WRITE (6,20) IDENT(IHEN),LAYD(IHEN),DIFEG,
     $                IFEED(IHEN),DIFFD,
     $                IWEIGH(IHEN),DIFWT
 20      FORMAT(2X,I4,5X,I4,4X,F8.1,5X,I4,4X,F8.1,
     $          6X,I4,4X,F8.1)
 25   CONTINUE
C
C  WRITE OUT TOTALS AND AVERAGES
C
      WRITE(6,30)ITOTEG,ITOTFD,ITOTWT
 30   FORMAT(//1X,'TOTALS:  ',I5,16X,I5,17X,I5)
      WRITE(6,35)AVEEG,AVEFD,AVEWT
 35   FORMAT(1X,'AVERAGE:  ',F6.1,15X,F6.1,16X,F6.1)
      STOP
      END
```

C.10 Chapter 12

Exercise 12A

```
C  Add M x N arrays A and B together and store in M x N
```

```
C  array C. Max M is 14, max N is 10.
C  This is just a series of statements, not a full program.

      REAL A(14,10), B(14,10), C(14,10)
      DO 100 I = 1,M
      DO 100 J = 1,N
         C(I,J) = A(I,J) + B (I,J)
100 CONTINUE
```

Exercise 12B

```
C  AUTHOR:  L  LANDAU
C  DATE  :  SEPT 1979
C  INPUT DESCRIPTION:
C        THERE IS NO INPUT
C
C  PURPOSE:
C        TO GENERATE THE NUMBERS 1 TO 30 IN AN ARRAY
C        AND TO WRITE OUT THE ARRAY
C        (A) ON ONE LINE
C        (B) SPREAD OVER 5 LINES WITH LINE NUMBERS
C
      INTEGER NUM(30)
C
C  PUT NUMBERS INTO NUM AND WRITE IT OUT
C
      DO 5 I = 1,30
         NUM(I) = I
  5   CONTINUE
      WRITE (6,10) (NUM(I),I=1,30)
 10   FORMAT(1X,'THE NUMBERS ARE'/
     +       '0',30(I3,1X))
C
C  NOW WRITE OUT THE ARRAY OVER 5 LINES
C PREFIXING EACH LINE WITH A LINE NUMBER
C
      IST = 1
      IFIN= IST+5
      DO 20 I = 1,5
         WRITE (6,15) I, (NUM(K), K = IST, IFIN)
 15      FORMAT(1X,I5,6I5)
         IST = IFIN+1
         IFIN= IFIN+6
 20   CONTINUE
      STOP
      END
```

Exercise 12C

```
C    AUTHOR:  LES LANDAU
C    DATE:    25TH MARCH 1981
```

```
C     INPUT DESCRIPTION:
                NO INPUT

C     PURPOSE:
C          TO GENERATE NUMBERS IN A TWO DIMENSIONAL ARRAY
C          OF SIZE 10 BY 3 AND TO WRITE OUT THE ARRAY:
C            (A) SPREAD OVER ONE LINE
C            (B) SPREAD OVER 5 LINES, WITH LINE NUMBERS


      INTEGER NUM (10,3)

C     PUT NUMBERS INTO THE ARRAY BY ROWS

      KNT = 1
      DO 20 I = 1,10
      DO 10 J = 1,3
         NUM (I,J) = KNT
         KNT = KNT + 1
 10   CONTINUE
 20   CONTINUE

C     NOW WRITE THEM OUT ON ONE LINE

      WRITE (6,30) ((NUM(I,J), J = 1,3), I = 1,10)
 30   FORMAT (1X, 'THE NUMBERS ARE'//
     $        1X, 30(I3,1X))

C     NOW WRITE THEM OUT OVER 5 LINES

      IR = 1
      DO 50 LINE = 1,5
         WRITE (6,40) LINE, ((NUM(I,J),J=1,3),I=IR,IR+1)
 40      FORMAT (1X,I5,6I5)
         IR = IR + 2
 50   CONTINUE
      STOP
      END
```

Exercise 12D

```
C  AUTHOR:  K. HANDEL
C  DATE:    5 AUGUST 1981
C  INPUT:   Rainfall data for 8 localities, one line per locality.
C           12 real numbers per line, in free format,
C           representing the rainfall for each of the 12 months
C           for a locality.
C  PURPOSE: Calculate average rainfall for each of 8 localities,
C           and also the entire rainfall figures for the month
C           with the highest average.
```

```
      REAL RAIN (8,12)
      DO 40 LOCAL = 1,8
         READ (5,*) (RAIN(LOCAL,MONTH),MONTH = 1,12)
C  Find the average rainfall for locality LOCAL.
         AVE = 0.0
         DO 20 MONTH = 1,12
            AVE = AVE + RAIN (LOCAL, MONTH)
20       CONTINUE
         WRITE (6,*) 'Average rainfall for locality', LOCAL,
     $               ' is', AVE/12.0
40    CONTINUE
C Find month with highest rainfall (HRAIN).
      HRAIN = -1
      DO 80 MONTH = 1,12
         TOT = 0.0
         DO 60 LOCAL = 1,8
            TOT = TOT + RAIN (LOCAL,MONTH)
60       CONTINUE
         IF (TOT .GT. HRAIN) THEN
            HRAIN = TOT
            MAXMTH = MONTH
         END IF
80    CONTINUE
      WRITE (6,*) ' Rainfall for month ',MAXMTH,', which had the '
     $  'highest average, was', (RAIN(LOCAL,MAXMTH), LOCAL = 1,8)
      STOP
      END
```

Exercise 12E

Algorithm

1. Initialize overall total
2. For each area,
   (a) initialize area total
   (b) for each month, read rainfall, and add to area total
   (c) find average for the area
   (d) add area total to overall total
3. Find overall average

Program

```
C Purpose : Read monthly rainfall figures and find the average
C for each area and the average over all areas.
C  Input : One monthly rainfall figure per line,
C         a real number in free format.
C         The order of input is : month 1 to month 12 for area 1,
C         up to   month 1 to month 12  for area 8.
C  Author : K. Handel   29-8-84
```

```
      INTEGER AREA
      TOTAL = 0
      DO 20 AREA = 1, 8
         ATOT = 0
         DO 10 MONTH = 1, 12
            READ (5,*) RAIN
            ATOT = ATOT + RAIN
10       CONTINUE
         AVE = ATOT / 12
         WRITE (6,*) 'Average for area',AREA,' is',AVE
         TOTAL = TOTAL + ATOT
20    CONTINUE
      AVE = TOTAL / (8 * 12)
      WRITE(6,*) 'Average over all areas is',AVE
      STOP
      END
```

Exercise 12F

```
C  AUTHOR:     K. HANDEL
C  DATE:       5 AUG 1981
C  INPUT:      LINE1 - an integer N, in free format, representing
C              the number of dimensions in space.
C              LINE2 - co-ordinates of a point in N-dimensional
C                      space, in free format.
C              Further lines each contain co-ordinates of a point.
C  PURPOSE:    Find the distance from the origin of points in
C              N-dimensional space. N is read from input, then
C              the co-ordinates of the points are read from input.
C  RESTRICTION: N, the number of dimensions in space, is less
C               than or equal to 25.
       REAL POS(25)
       READ (5,*) N
       IF (N.GT.25) THEN
          WRITE (6,*) 'Max number of dimensions is 25. ',
     $                'Number read is',N
          STOP
       END IF
  100  READ (5,*,END=140) (POS(I),I = 1,N)
       DIST = 0
       DO 120 I = 1,N
          DIST = DIST + POS(I)**2
  120  CONTINUE
       WRITE (6,*) 'Distance of (',(POS(I),I = 1,N), ') ',
     $             'from origin is ', SQRT(DIST)
       GO TO 100
  140  STOP
       END
```

C.11 Chapter 13

Exercise 13A

1. ok
2. ok
3. illegal (mismatch of parameter types)
4. illegal (wrong number of parameters)
5. illegal (mismatch of parameter types)
6. this would work, but confusing and should not be used.
7. legal, but could cause an error accessing out of bounds in arrays ct and sunny.

Exercise 13B

```
C  PURPOSE: Find the average and number of zeros in a
C           set of integers which are read from input.
C  INPUT   : Line 1 contains the number of integers to follow.
C           Following line(s) contain the integers in free format.
      INTEGER JA(50)
      READ (5,*) N
      IF (N .GT. 50) THEN
         WRITE (6,*) N, ' is too big'
         STOP
      END IF

      READ (5,*) (JA(I), I = 1, N)

      CALL AVERNZ (JA, N, AVER, NZ)

      WRITE (6,*) 'The average is', AVER
      WRITE (6,*) 'The number of zeros is', NZ
      STOP
      END

      SUBROUTINE AVERNZ(JA,N,AVER,NZ)
C  AUTHOR:       K. HANDEL
C  DATE:         5 AUG 1981
C  INPUT:        None
C  PURPOSE:      Find the average of the first N elements of
C                an integer array and the number of those
C                elements which are zero.
C  PARAMETERS:
C           INPUT:    JA - array of length N (INTEGER)
C                      N - dimension of JA  (INTEGER)
C          OUTPUT:  AVER - average of first N elements
C                          of JA (REAL)
C                     NZ - number of zeros in first N
C                          elements of JA (INTEGER)
```

```
      INTEGER JA(N)
      NZ = 0
      AVER = 0.0
      DO 10 I = 1,N
         AVER = AVER + JA(I)
         IF (JA(I) .EQ. 0) NZ = NZ + 1
   10 CONTINUE
      AVER = AVER/N
      RETURN
      END
```

Exercise 13C

```
      REAL A(20,15)
      MAXROW = 20
      MAXCOL = 15
C
C  Read size of array required
      READ (5,*) NROWS, NCOLS

      CALL READA (A, NROWS,NCOLS, MAXROW,MAXCOL)
      ...
      SUBROUTINE READA (A, NROWS,NCOLS, MAXROW,MAXCOL)
C  AUTHOR:       K.HANDEL
C  DATE:         5 AUG 1981
C  INPUT:        NROWS lines of data, each containing
C                NCOLS real numbers in free format.
C  PURPOSE:      read data into array A.
C  PARAMETERS:
C       INPUT:  NROWS = number of rows to be read
C               NCOLS = number of columns to be read
C               MAXROW = no. of rows in A
C               MAXCOL = no. of cols in A
C      OUTPUT:  A     = real array, dimensions MAXROW x MAXCOL.
      REAL A(MAXROW,MAXCOL)
      IF (NROWS .GT. MAXROW .OR. NCOLS .GT. MAXCOL) THEN
         WRITE (6,*) Dimensions are too big. Max size is',
     $               MAXROW, MAXCOL
         STOP
      END IF

      DO 10 IROW = 1,NROWS
          READ (5,*) (A(IROW,ICOL), ICOL = 1,NCOLS)
   10 CONTINUE
      RETURN
      END
```

Exercise 13D

```
C  AUTHOR:       K. HANDEL
C  DATE:         29 AUG 1984
```

```
C  INPUT:  (A) the number of numbers to be amended - an even integer,
C              right-justified in columns 1-5.
C          (B) the numbers to be amended - starting in column 1
C              on a new line, put 5 real numbers per line,
C              10 columns per number. If a number does not
C              have a decimal point, 3 decimal places will be
C              assumed. [ Read in subroutine INPUT ]
C  PURPOSE:    Read a list of real numbers, and amend them so that
C              every odd-numbered element in the list is replaced
C              by the sum of itself and the next element; and every
C              even-numbered element is replaced by the absolute
C              difference between itself and the previous element.
C  The following subroutines are used:
C      INPUT(ALIST,N) - reads N real numbers into array ALIST.
C     OUTPUT(ALIST,N) - prints the first N elements of ALIST.
C      AMEND(ALIST,N) - amends the first N elements of ALIST
C RESTRICTIONS:  ALIST contains real numbers. To amend a list
C                of integers, change the declarations in the
C                main program and all subroutines.
C                A max of 100 numbers may be amended. To
C                increase this maximum, change the
C                declaration in the main progam.
      REAL ALIST(100)
      READ (5,10) N
   10 FORMAT(I5)
      IF (N .GT. 100) THEN
         WRITE (6,*) N, ' is too big. Max is 100'
         STOP
      END IF
      IF ( MOD(N,2) .NE. 0 ) THEN
         WRITE (6,*) 'Can''t operate on an odd number of numbers!'
         STOP
      END IF

      CALL INPUT (ALIST,N)
      WRITE (6,20)
   20 FORMAT ('1Original data'///)
      CALL OUTPUT (ALIST,N)
      CALL AMEND (ALIST,N)
      WRITE (6,30)
   30 FORMAT('1Amended data'///)
      CALL OUTPUT (ALIST,N)
      STOP
      END
      SUBROUTINE INPUT (A,N)

C  PURPOSE: Read N real numbers into array A, in 5F10.3 format
C  PARAMETERS: INPUT - N = number of integers (INTEGER)
C              OUTPUT - A = array of N numbers (REAL)
      REAL A(N)
      READ (5,10) (A(I),I = 1,N)
```

```
   10 FORMAT (5F10.3)
      RETURN
      END
      SUBROUTINE OUTPUT (A,N)
C  PURPOSE: Print the first N numbers from array A,
C           double spaced.
C  PARAMETERS:  INPUT - N = number of numbers (INTEGER)
C                       A = array of N numbers (REAL)
      REAL A(N)
      WRITE (6,10) (A(I),I = 1,N)
   10 FORMAT ('0',5F10.3)
      RETURN
      END
      SUBROUTINE AMEND (A,N)
C  PURPOSE: Amend the first N elements of array A.
C  PARAMETERS: INPUT - N = number of elements to be amended
C                          (an even INTEGER).
C                      A = original list (REAL)
C             OUTPUT - A = amended list
      REAL A(N), ODD, EVEN
      DO 40 I = 1, N-1, 2
         ODD    = A(I)
         EVEN   = A(I+1)
         A(I)   = ODD + EVEN
         A(I+1) = ABS (ODD - EVEN)
   40 CONTINUE
      RETURN
      END
```

C.12 Chapter 14

Exercise 14A

```
      FUNCTION AREA (R)
C  PURPOSE:  to calculate the area of a circle of radius R
C  PARAMETERS:
C    NAME         TYPE      DESCRIPTION
C     R           REAL      radius of the circle
      PI = 4.0*ATAN(1.0)
      AREA = PI * R**2
      RETURN
      END
```

Exercise 14B

```
      FUNCTION FN (X)

C  PARAMETERS:
C     NAME      TYPE  DESCRIPTION
C      X        REAL  A PARAMETER TO THE FUNCTION
C  PURPOSE:
```

```
C       TO CALCULATE THE EXPRESSION
C            2                  2   1/2
C          X  + (1 + 2X + 3X  )
C
C        A  WARNING MESSAGE WILL BE SENT IF THE SQUARE ROOT
C        CANNOT BE TAKEN (DUE TO NEGATIVE ARGUMENT). IN THIS
C        CASE A VALUE OF ZERO WILL BE RETURNED BY THE FUNCTION.

      VAL = 1 + 2*X +3*X**2
      IF (VAL .LT. 0) THEN
         WRITE (6,*) 'ROOT NEGATIVE FOR VALUE OF X OF ', X,
     $               ' ----ZERO RETURNED----> ERROR!!'
         FN = 0.0
      ELSE
         FN = X**2 + SQRT (VAL)
      END IF
      RETURN
      END
```

USE OF THE FUNCTION WOULD BE

1. `A = (6.9 + Y) / FN (Y)`
2. `B = (2.1*Z + Z**4) / FN(Z)`
3. `C = SIN(Y) / FN(Y**2)`
4. `D = 1.0 / FN ( SIN(Y) )`

Exercise 14C

```
      FUNCTION AREA (R,P)
C  AUTHOR:     K. HANDEL
C  DATE        5 AUG 1981
C  INPUT:      None
C  PURPOSE:   If P = 1, calculate area of an equilateral
C                        triangle of side R.
C            If P = 2, calculate area of a square of side R.
C            If P = 3, calculate area of a circle of radius R.
C            Return the result as the function value (REAL).
C  PARAMETERS : INPUT - P = switch to indicate which type of area
C                           is required (INTEGER)
C                       R = length of side or radius (REAL)
      INTEGER P
      REAL R, RSQ
      AREA = 0.0
      RSQ = R * R

C  EQUILATERAL TRIANGLE

      IF (P .EQ. 1) AREA = SQRT (3.0) * RSQ/4.0

C  SQUARE
```

```
      IF (P .EQ. 2) AREA = RSQ

C  CIRCLE

      IF (P .EQ. 3) AREA = 3.14159 * RSQ
      RETURN
      END
```

Exercise 14D

```
      FUNCTION  SUM (A,M,N)

C  AUTHOR:      K. HANDEL
C  DATE:        11 AUGUST 1981
C  INPUT:       None
C  PURPOSE:     Return the sum of the absolute values of all
C               elements in M x N array A.
C  PARAMETERS: A - M x N array (REAL)
C              M,N - integer dimensions of array A  (ROWS,COLUMNS)
C  NO PARAMETER IS ALTERED IN THE FUNCTION.
      REAL A(M,N)
      SUM = 0.0
      DO 100 I = 1,M
      DO 100 J = 1,N
         SUM = SUM + ABS(A(I,J))
  100 CONTINUE
      RETURN
      END
```

Note that this answer assumes that A is declared to be exactly M by N in the calling program. If A had been declared as MAXM by MAXN but only the first M rows and N columns were to be used by the function, then the function header would be

```
      FUNCTION NZ (IARRAY, M,N, MAXM,MAXN)
```

and the array declaration in the function would be

```
      INTEGER IARRAY(MAXM,MAXN)
```

Exercise 14E

```
      FUNCTION  AMAX (A,N)
C  AUTHOR:     K. HANDEL
C  DATE:       11 AUG 1981
C  INPUT:      None
C  PURPOSE:    Return the max value in array A.
C  PARAMETERS:  A - 1-dimensional real array of length N.
C               N - integer dimension of A.
C  NO PARAMETER IS ALTERED IN THE FUNCTION.
      REAL  A(N)
      AMAX = A(1)
      DO 100 I = 2,N
```

```
         IF (A(I) .GT. AMAX) AMAX = A(I)
  100 CONTINUE
      RETURN
      END
```

Exercise 14F

```
      FUNCTION  NZ (IARRAY,M,N)

C  AUTHOR:    K. HANDEL
C  DATE:      11 AUGUST 1981
C  INPUT:     None
C  PURPOSE:   Return a count of the number of zeros
C             in M x N array IARRAY.
C  PARAMETERS:  IARRAY - integer array with dimensions M x N.
C                 M   - number of rows in IARRAY.
C                 N   - number of columns in IARRAY.
C  NO PARAMETER IS ALTERED IN THE FUNCTION.
      INTEGER IARRAY(M,N)
      NZ = 0
      DO 100 I = 1,M
      DO 100 J = 1,N
         IF (IARRAY(I,J) .EQ. 0) NZ = NZ + 1
  100 CONTINUE
      RETURN
      END
```

Exercise 14G

```
      FUNCTION NFACT(N)
C  N! overflows for N > 12.
      IF (N .GT. 12) THEN
         WRITE (6,*) N, ' is too big. 12 is max.'
         NFACT = 0
         RETURN
      END IF
      NFACT = 1
      DO 25 I = 2, N
         NFACT = NFACT*I
   25 CONTINUE
      RETURN
      END
```

Exercise 14H

```
C                n            n          n!
C    Calculates  C    where   C    =  --------
C                 r            r      r!(n-r)!
C    for non-negative integers n and r.
C    Input : n and r in free format.
C    K. Handel   28-8-84
```

```
      INTEGER R, RFACT
      READ(5,*) N, R
      IF (N.LT.0 .OR. R.LT.0 .OR. N.LT.R) THEN
         WRITE(6,*) 'I can''t do that'
         STOP
      END IF
      NCR = NFACT(N)/NFACT(R)/NFACT(N-R)
      WRITE(6,*)'There are ',NCR,' ways of choosing ',R,
     $ 'objects from ', N, 'objects'
      STOP
      END
C
      FUNCTION NFACT(N)
C  N! overflows for N > 12.
      IF (N .GT. 12) THEN
         WRITE (6,*) N, ' is too big. 12 is max.'
         NFACT = 0
         RETURN
      END IF
      NFACT = 1
      DO 25 I = 2, N
         NFACT = NFACT*I
 25   CONTINUE
      RETURN
      END
```

C.13 Chapter 15

Exercise 15A

```
C  AUTHOR:       K.  HANDEL
C  DATE:         11 AUG 1981
C  INPUT:        Lines of text. End of data is end-of-file marker.
C  PURPOSE:      For each line of text read in, print HAPPY BIRTHDAY
C                followed by the text.
      CHARACTER TEXT*80, HB*15
      HB = 'HAPPY BIRTHDAY'
   10 READ (5,20,END = 40) TEXT
   20 FORMAT (A)
      WRITE (6,30) HB,TEXT
   30 FORMAT (2(1X,A))
      GO TO 10
   40 STOP
      END
```

Exercise 15B

```
C     AUTHOR:  LES LANDAU

C     DATE:  5TH JULY 1981
```

```
C                (NOT INCLUDING THE FOLLOWING DELIMITER)

      LEN = 0
      DELIM (1) = '.'
      DELIM (2) = ','
      DELIM (3) = ' '
      NDEL      = 3
      IF (ISTART .GT. 80 .OR. ISTART .LT. 1) THEN
          LEN = 0
          RETURN
      END IF

C   LOOK FOR A WORD STARTING FROM ISTART, JUMP OUT OF
C   THE LOOP WHEN A DELIMITER HAS BEEN FOUND

      DO 50 LOOK = ISTART,80
      DO 40 J    = 1,NDEL
         IF (LINE (LOOK:LOOK) .EQ. DELIM(J)) GO TO 60
 40   CONTINUE
 50   CONTINUE

C  COME HERE IF NO DELIMITER HAS BEEN FOUND

      WORD = LINE (ISTART:)
      LEN = 80 - ISTART + 1
      RETURN

C    COME HERE WHEN A DELIMITER HAS BEEN FOUND

 60   CONTINUE
      WORD = LINE(ISTART:LOOK)
      LEN = LOOK - ISTART
      RETURN
      END
```

Exercise 15C

```
C  AUTHOR:     K. HANDEL
C  DATE:       11 AUG 1981
C  INPUT:      An integer in the range 1-12, repesenting a month.
C  PURPOSE:    Given a month represented by an integer, print the
C              name of the month (abbreviated to 3 characters).
      CHARACTER*3 MONTH(12)
      DATA MONTH/'JAN','FEB','MAR','APR','MAY','JUN',
     $           'JUL','AUG','SEP','OCT','NOV','DEC'/
      READ (5,*) I
      IF (I .GE. 1 .AND. I .LE. 12) WRITE (6,10) MONTH(I)
   10 FORMAT (1X,A3)
      STOP
      END
```

Chapter 16 describes the DATA statement, which is the best way of setting up MONTH.

Exercise 15D

```
C  AUTHOR:       K. HANDEL
C  DATE:         11 AUG 1981
C  INPUT:        An integer in the range 1-7, representing a day
C                of the week.
C  PURPOSE:      Given a day represented by an integer, print
C                the day (in characters).
      CHARACTER*9  DAY(7)
      DATA DAY/'MONDAY   ','TUESDAY  ','WEDNESDAY','THURSDAY ',
     $         'FRIDAY   ','SATURDAY ','SUNDAY   '/
      READ (5,*) J
      IF (J .GE. 1 .AND. J .LE. 7)  WRITE (6,10) DAY (I)
   10 FORMAT (1X,A)
      STOP
      END
```

Note: The DATA statement is described in Chapter 16.

Exercise 15E

```
C  AUTHOR:     K. HANDEL
C  DATE:       11 AUG 1981
C  INPUT: (A) Product information for a company.
C              The first line contains the number of products
C              (INTEGER in free format).
C              Then put information on each product on a separate
C              line as follows:-
C              columns 1-5  product code (5 characters)
C              columns 11-3  product description (20 characters)
C              Quotes should not be put around codes or
C              descriptions.
C         (B) Product codes for which descriptions are required.
C              Any number of product codes may be read, one per line
C              in columns 1-5. Input is terminated by end-of-file.
C  PURPOSE:    Read all product information, then for given
C              product codes, print the corresponding descriptions.
C  SUBROUTINES CALLED:  DSPLAY - prints code and description for
C                                a given code.
      CHARACTER CODE*5(50), DESCR*20(50), ACODE*5
C  Read the number of products (N) then read codes and
C  descriptions.
      READ (5,*) N
      IF (N .GT. 50) THEN
         WRITE (6,*) 'No more than 50 products are allowed. N = ',N
         STOP
      END IF
      DO 20 I = 1,N
         READ (5,10) CODE(I), DESCR(I)
```

```
   10    FORMAT (A5,5X,A20)
   20 CONTINUE
C  Print heading for next section
      WRITE (6,25)
   25 FORMAT ('1CODE       DESCRIPTION'//)
C  Read product codes for which descriptions are required.
C  Call DSPLAY to print descriptions.
   30 READ(5,40,END = 50) ACODE
   40 FORMAT (A5)
      CALL DSPLAY (ACODE,CODE,DESCR,N)
      GO TO 30
   50 STOP
      END

      SUBROUTINE DSPLAY (ACODE,CODE,DESCR,N)
C  PARAMETERS:  ACODE - a 5-character product code
C                CODE - a 1-dimensional array of length N,
C                       containing 5-character product codes.
C               DESCR - a 1-dimensional array of length N,
C               containing 20-character product descriptions.
C                   N - integer dimension of arrays CODE and DESCR.
C            NO PARAMETERS ARE ALTERED IN THE SUBROUTINE.
C  PURPOSE:  Search for ACODE in the list of codes in CODE,
C            then print it and its corresponding description.
C            A linear search is used.
      CHARACTER*5 ACODE, CODE(N), DESCR*20 (N)
      DO 10 I = 1,N
         IF (ACODE .EQ. CODE(I)) GO TO 20
   10 CONTINUE
      WRITE (6,*) 'CODE ',ACODE, ' NOT FOUND'
      RETURN
   20 WRITE (6,30) ACODE, DESCR (I)
   30 FORMAT (1X,A5,5X,A20)
      RETURN
      END
```

Exercise 15F

```
C  AUTHOR:   K. Handel
C  DATE:     11 Aug 1981
C            corrected 22 Apr 1982
C  PURPOSE:  Print the day and date in full for each day of 1981
C            read in. Eg. if input is 328, output is
C            328     TUESDAY   24th NOVEMBER  1981

C  INPUT:    Integers repesenting days of the year, one per line,
C            in free format. Input is terminated by end-of-file.
C  RESTRICTIONS:  only works for 1981 because it uses the facts
C                 that Jan 1, 1981  is a Thursday, and 1981 is
C                 not a leap year.
      CHARACTER  MONTH*9(12), DAY*9(7), TH*2(4)
```

```
      INTEGER MTHEND(0:11), YEAR
      DATA  MONTH/'JANUARY  ','FEBRUARY ','MARCH    ','APRIL    ',
     $            'MAY      ','JUNE     ','JULY     ','AUGUST   ',
     $            'SEPTEMBER','OCTOBER  ','NOVEMBER ','DECEMBER '/
      DATA  DAY/'MONDAY   ','TUESDAY  ','WEDNESDAY','THURSDAY ',
     $          'FRIDAY   ','SATURDAY ','SUNDAY   '/
      DATA  TH/'st','nd','rd','th'/, YEAR/1981/
C  Day of year of last day in each month, for a non-leap year:
      DATA  MTHEND/0,31,59,90,120,151,181,212,243,273,304,334/
C  Read a day of the year.
   10 READ (5,*,END = 50) IDATE
      IF (IDATE .LE. 0 .OR. IDATE .GT. 365) THEN
         WRITE (6,*) IDATE, ' is an illegal date. Ignored.'
         GO TO 10
      END IF
C  Find the month, M.
      DO 20 M = 1,11
         IF (IDATE .LE. MTHEND(M)) GO TO 30
   20 CONTINUE
C  Find the day of month, MDAY, and appropriate suffix, TH(ITH).
   30 MDAY = IDATE - MTHEND(M - 1)
      ITH = MOD(MDAY,10)
      IF (ITH .EQ. 0 .OR. ITH .GE. 4 .OR. MDAY .GE. 11 .AND.
     $    MDAY .LE. 13) ITH = 4
C  Find the day of week, IDAY.
C  Jan 1, 1981  is Thursday, ie. 4th day, so add 3 to IDATE.
      IDAY = MOD (IDATE + 3, 7)
      IF (IDAY .EQ. 0) IDAY = 7
C  Print the lot and read another date.
      WRITE (6,40) IDATE, DAY(IDAY), MDAY, TH(ITH), MONTH(M), YEAR
   40 FORMAT (20X, I5, 5X, A, 1X, I2, A, 1X, A, I5)
      GO TO 10
   50 STOP
      END
```

Note: The DATA statement is described in Chapter 16.

Exercise 15G

```
C  AUTHOR:   K.HANDEL
C  DATE:     11 AUG 1981
C  INPUT:    Lines of text containing only digits, commas,
C            plus and minus. Input is terminated by end-of-file.
C  PURPOSE:  Decode the text into integers, and print both the
C            lines of text and the integer values in the text.
C  FUNCTION CALLED:
C      LENGTH - finds the length of TEXT, excluding
C               blanks at end of line.
      CHARACTER  TEXT*80, CH*1, SIGN*1
   10 READ (5,20,END = 50) TEXT
   20 FORMAT (A)
```

```
      WRITE (6,30) TEXT
   30 FORMAT (//1X,A)
      L = LENGTH (TEXT)
      IF (L .EQ. 0) GO TO 10
C  initialize
      SIGN = '+'
      NUM = 0
      DO 40 I = 1,L
         CH = TEXT (I:I)
         IF (CH .EQ. '+' .OR. CH .EQ. '-') THEN
            SIGN = CH
         ELSE IF (CH .GE. '0' .AND. CH .LE. '9') THEN
            NUM = NUM*10 + ICHAR(CH) - ICHAR('0')
         ELSE IF (CH .EQ. ',') THEN
C           correct sign, print and reset number
            IF (SIGN .EQ. '-') NUM = -NUM
            WRITE (6,*) NUM
            NUM = 0
            SIGN = '+'
         ELSE
            WRITE (6,*) CH, ' is an illegal character. Ignored.'
         END IF
   40 CONTINUE
C  Print last number in TEXT unless it was terminated by a comma.
      IF (CH .NE. ',') THEN
         IF (SIGN .EQ. '-') NUM = -NUM
         WRITE (6,*) NUM
      END IF
      GO TO 10
   50 STOP
      END
      FUNCTION LENGTH (TEXT)
C  PURPOSE:  Return length of TEXT when blanks at end are
C            excluded.
      CHARACTER*80 TEXT
      DO 60 I = 80,1,-1
         IF (TEXT(I:I) .NE. ' ') GO TO 70
   60 CONTINUE
   70 LENGTH = I
      RETURN
      END
```