

1. Fortran Background

- 1.1 Fortran history and standards
- 1.2 Source code and executable code
- 1.3 Fortran 95 Compilers

2. Running a Fortran program

- 2.1 NAG Fortran – using the FBuilder IDE
- 2.2 NAG Fortran – using the command window
- 2.3 Salford Compiler – using the Plato IDE
- 2.4 Salford Fortran – using the command window
- 2.5 Irritating features of the University PCs
- 2.6 Getting help

3. A simple program**4. Basic elements of Fortran**

- 4.1 Variable names
- 4.2 Data types
- 4.3 Declaration of variables
- 4.4 Numeric operators and expressions
- 4.5 Character operators
- 4.6 Logical operators and expressions
- 4.7 Line discipline
- 4.8 Miscellaneous remarks

5. Repetition: DO and DO WHILE

- 5.1 Types of DO loop
- 5.2 Deterministic DO loops
- 5.3 Non-deterministic DO loops
- 5.4 Nested DO Loops
- 5.5 Non-Integer Steps

6. Decision-making: IF and CASE

- 6.1 The IF construct
- 6.2 The CASE construct

7. Arrays

- 7.1 One-dimensional arrays (vectors)
- 7.2 Array declaration
- 7.3 Dynamic arrays
- 7.4 Array input/output and implied DO loops
- 7.5 Array-handling functions
- 7.6 Element-by-element operations
- 7.7 Matrices and higher-dimension arrays
- 7.8 Array initialisation
- 7.9 Array assignment and array expressions
- 7.10 The WHERE construct

8. Character handling

- 8.1 Character constants and variables
- 8.2 Character assignment
- 8.3 Character operators
- 8.4 Character substrings
- 8.5 Comparing and ordering
- 8.6 Character-handling functions

9. Functions and subroutines

- 9.1 Intrinsic subprograms
- 9.2 Program units
- 9.3 Subprogram arguments
- 9.4 The SAVE attribute
- 9.5 Array arguments
- 9.6 Character arguments

10. Advanced input/output

- 10.1 READ and WRITE
- 10.2 Input/output with files
- 10.3 Formatted WRITE
- 10.4 The READ statement
- 10.5 Repositioning input files
- 10.6 Additional specifiers

11. Modules

- 11.1 Sharing variables
- 11.2 Internal functions
- 11.3 Example
- 11.4 Compiling programs with modules

12. Other Fortran Features

- 12.1 Derived data types
- 12.2 Pointers
- 12.3 Object-oriented programming (Fortran 2003)

Appendices

- A1. Order of statements in a program unit
- A2. Fortran statements
- A3. Type declarations
- A4. Intrinsic routines
- A5. Operators

Recommended Books

- Hahn, B.D., 1994, *Fortran 90 For Scientists and Engineers*, Arnold
Chapman, S.J., 2007, *Fortran 95/2003 For Scientists and Engineers* (2nd Ed.), McGraw-Hill

1. FORTRAN BACKGROUND

1.1 Fortran History and Standards

Fortran (FORMula TRANslation) was the first high-level programming language. It was devised by John Bachus in 1953. The first Fortran compiler was produced in 1957.

Fortran is highly *standardised*, making it extremely *portable* (able to run on a wide range of computers and operating systems). It is an evolving language, passing through a sequence of international standards:

Fortran 66 – the original ANSI standard (accepted 1972!);

Fortran 77 – ANSI X3.9-1978 – structured programming;

Fortran 90 – ISO/IEC 1539:1991 – major revision, modernises Fortran;

Fortran 95 – ISO/IEC 1539-1: 1997 – very minor revision;

Fortran 2003 – ISO/IEC 1539-1:2004(E) – adds object-oriented support and interoperability with C

Many Fortran compilers exist. The Fortran 95 compilers available in the University clusters are:

- Silverfrost (formerly, Salford¹) compiler, FTN95
- NAG compiler, nagfor.

1.2 Source Code and Executable Code

For all high-level languages (Fortran, C, Pascal, ...) programmes are written in *source code*. This is a human-readable set of instructions that can be created or modified on any computer with any text editor. Filetypes identify the programming language; e.g.

Fortran 95 files typically have type .f95

C++ files typically have type .cpp

The job of the *compiler* is to turn this into machine-readable *executable* code.

Under Windows, executable files have type .exe

In this course the programs are very simple and will usually be contained in a single .f95 file. However,

- in real engineering problems code is often contained in many separate source files;
- producing executable code is actually a two-stage process:
 - *compiling* converts each individual source file into intermediate *object* code;
 - *linking* combines all the object files with additional library routines to create an *executable* program.

Most Fortran code consist of multiple *subprograms*, all performing specific, independent functions. These may be in one file or in separate files. The latter arrangement allows them to be re-used in different applications.

1.3 Fortran 95 Compilers

There are many Fortran 95 compilers. Free (with restrictions) Fortran 95 compilers for Windows include:

Silverfrost FTN95 <http://www.silverfrost.com/> – an upgraded version of the compiler used here.

G95 <http://www.g95.org/>

Gnu Fortran <http://gcc.gnu.org/wiki/GFortran>

The other Fortran compilers available in the University for use with Windows are:

NAG Fortran <http://www.nag.co.uk/nagware/np.asp> - available in the clusters

Intel Fortran Have to talk nicely to IT services about this.

If you prefer the linux operating system to Windows then the gfortran compiler is also available on the PCs which allow a choice of operating systems.

The web resources for this course contain a long list of commercial Fortran compilers.

¹ Salford Software has now passed on any future marketing and development of this product to a software company called Silverfrost, so on the web you will see it referred to as Silverfrost FTN95.

2. RUNNING A FORTRAN PROGRAM

You may create, edit, compile and run a Fortran program either:

- from the command line;
- in an Integrated Development Environment (IDE).

Whatever you choose to compile and run the code you are quite entitled to create the source code with any editor you like: for example, notepad.

Create the following source file and call it `prog1.f95`.

```
PROGRAM HELLO
  PRINT *, 'Hello, world!'
END PROGRAM HELLO
```

Compile and run this code using any of the methods in 2.1 – 2.4.

2.1 NAG Fortran – Using the FBuilder IDE

Start the IDE:

```
Start - All Programs - Site-licensed applications - Compilers
      - NAG Fortran Compiler 5.2
```

Either open the pre-existing source file or create it (and save it) using the IDE's built-in editor.

Run it from the "Execute" icon on the toolbar. This will automatically compile, then run. An executable `prog1.exe` will appear in the same folder as the source file.

2.2 NAG Fortran – Using the Command Window

Open a command window:

```
Start - Run - cmd      (or just look in the standard Windows accessories)
```

Navigate to any suitable folder; e.g.

```
cd \work
```

Create (and then save) the source code

```
notepad prog1.f95
```

Compile the code by entering

```
nagfor prog1.f95
```

or, if you prefer the executable to be called `prog1.exe` rather than the dull default `a.exe`:

```
nagfor -o prog1 prog1.f95
```

Run the executable by typing its name.

2.3 Salford Compiler – Using the Plato IDE

Start the IDE:

```
Start - All Programs - Site-licensed applications - Compilers
      - Salford - Plato
```

Either open the pre-existing source file or create it (and save it as a `.f95` file) using the IDE's own editor.

Run it from the "Execute" icon on the toolbar. This will automatically compile, then run. An executable `prog1.exe` will appear in the same folder as the source file.

2.4 Salford Fortran – Using the Command Window

Open a command window:

```
Start - Run - cmd      (or just look in the standard Windows accessories)
```

Navigate to any suitable folder; e.g.

```
cd \work
```

Create (and then save) the source code

```
notepad prog1.f95
```

Compile and link the code by entering

```
ftn95 prog1.f95 /link
```

This will automatically create an executable `prog1.exe`. (If you don't include the `/link` option it will only create an object file `prog1.obj`.)

Run the executable by entering

```
prog1
```

2.5 Irritating Features of the University PCs

It is possible that the operating system's `PATH` environment variable will not be set to find the relevant software. In this case you *must* open the command window from the compiler's group on the `Start` menu, not just `run - cmd`. (Entirely at the whim of IT Systems, this problem could occur any time.)

The software packages will have to be downloaded for first use on a particular PC. This occurs automatically, but it might take some time. There is now a nice café in the Pariser building.

If the software has just been downloaded then you may need a reboot.

2.6 Getting Help

Help (for both the Fortran language and the use of the compiler) is available from pull-down menus in both `Fbuilder` and `Plato` IDEs.

Help is also available from the command line:

```
nagfor -help      (compiler options only)
```

or

```
ftn95 /help
```

3. A SIMPLE PROGRAM

Example. Quadratic-equation solver (real roots).

The well-known solutions of the quadratic equation

$$Ax^2 + Bx + C = 0$$

are

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

The roots are real if and only if the *discriminant* $B^2 - 4AC$ is greater than or equal to zero.

A program which asks for the coefficients and then outputs the real roots might look like the following.

```
PROGRAM ROOTS
! Program solves the quadratic equation Ax**2+Bx+C=0
  IMPLICIT NONE
  REAL A, B, C                               ! declare variables
  REAL DISCRIMINANT, ROOT1, ROOT2

  PRINT *, 'Input A, B, C'                   ! request coefficients
  READ *, A, B, C

  DISCRIMINANT = B ** 2 - 4.0 * A * C        ! calculate discriminant

  IF ( DISCRIMINANT < 0.0 ) THEN
    PRINT *, 'No real roots'
  ELSE
    ! Calculate roots
    ROOT1 = ( -B + SQRT( DISCRIMINANT ) ) / ( 2.0 * A )
    ROOT2 = ( -B - SQRT( DISCRIMINANT ) ) / ( 2.0 * A )
    PRINT *, 'Roots are ', ROOT1, ROOT2     ! output roots
  END IF
END PROGRAM ROOTS
```

This example illustrates many of the features of Fortran.

(1) Statements

Fortran source code consists of a series of *statements*. The usual use is one per line (interspersed with blank lines for clarity). However, we shall see later that it is possible to have more than one statement per line and for one statement to run over several lines.

Lines may be up to 132 characters long.

(2) Comments

The exclamation mark (!) signifies that everything after it on that line is a *comment* (i.e. ignored by the compiler, but there for your information). Sprinkle liberally.

(3) Constants

Elements whose values don't change are termed *constants*. Here, 2.0 and 4.0 are *numerical constants*. The presence of the decimal point indicates that they are of *real* type. We shall discuss the difference between real and integer types later.

(4) Variables

Entities whose values can change are termed *variables*. Each has a *name* that is, basically, a symbolic label associated with a specific location in memory. To make the code more readable, names should be descriptive and meaningful; e.g. DISCRIMINANT in the above example.

All the variables in the above example have been declared of *type* REAL. Other types (INTEGER, CHARACTER, LOGICAL, ...) will be introduced later, where we will also explain the IMPLICIT NONE statement.

Variables are *declared* when memory is set aside for them by specifying their type, and *defined* when some value is assigned to them.

(5) Operators

Fortran makes use of the usual *binary numerical* operators +, -, * and / for addition, subtraction, multiplication and division, respectively. ** indicates exponentiation ('to the power of').

Note that '=' is an *assignment operation*, not a mathematical equality. Read it as "becomes". It is perfectly legitimate (and, indeed, common practice) to write statements like

```
N = N + 1
```

meaning "add 1 to variable N"

(6) Intrinsic Functions

The Fortran standard provides some *intrinsic* (that is, built-in) functions to perform important mathematical functions. The square-root function SQRT is used in the example above. Others include COS, SIN, LOG, EXP, TANH. A list of useful mathematical intrinsic functions is given in Appendix A4.

Note that, in common with all other scientific programming languages, the trigonometric functions SIN, COS, etc. expect their arguments to be in *radians*.

(7) Simple Input/Output

Simple *list-directed* input and output is achieved by the statements

```
READ *, list  
PRINT *, list
```

respectively. The contents are determined by what is in *list* and the *'s indicate that the input is from the keyboard and that the computer should decide how to format the output. Data is read from the *standard input device* (usually the keyboard) and output to the *standard output device* (usually the screen). In Section 10 it will be shown how to read from and write to files and how to produce *formatted* output.

(8) Decision-making

All programming languages have some facility for decision-making: doing one thing if some condition is true and (optionally) doing something else if it is not. The particular form used here is

```
IF ( some condition ) THEN  
    [ do something ]  
ELSE  
    [ do something else ]  
END IF
```

We shall encounter various other forms of the IF construct in Section 6.

(9) The PROGRAM and END PROGRAM statements

Every Fortran program has one and only one *main program*. We shall see later that it can have many *subprograms* (*subroutines* or *functions*). The main program has the structure

```
[ PROGRAM [ name ] ]  
    [ declaration statements ]  
    [ executable statements ]  
END [ PROGRAM [ name ] ]
```

Everything in square brackets [] is optional. However, it is good programming practice to put the name of the program in both header and END statements, as in the quadratic-equation example above.

(10) Cases and Spaces

Except within character strings, Fortran is completely *case-insensitive*. Everything may be written in upper case, lower case or a combination of both, and we can refer to the same variable as ROOT1 and root1 within the same program unit. *Warning*: this is a very bad habit to get into, because it is not true in some major programming languages: notably C and C++.

Spaces are generally valid everywhere except in the middle of names and keywords. As with comments, they should be sprinkled liberally to aid clarity.

Indentation is optional but is highly recommended because it makes it much easier to understand a program's structure. It is common to indent a program's contents by 2 or 3 spaces from its header and END statements, and to indent the statements contained within, for example, IF constructs or DO loops by a similar amount.

(11) Running the Program.

Follow the instructions in Section 2 to compile and link the program.

Run it by entering its name at the command prompt or from within an IDE. It will ask you for the three coefficients A, B and C.

Try A=1, B=3, C=2 (i.e. $x^2 + 3x + 2 = 0$). The roots should be -1 and -2. You can input the numbers as

```
1 3 2 [enter]
```

or

```
1, 3, 2 [enter]
```

or even

```
1 [enter]
```

```
3 [enter]
```

```
2 [enter]
```

Now try the combinations

A = 1, B = -5, C = 6

A = 1, B = -5, C = 10 (What are the roots of the quadratic equation in this case?)

4. BASIC ELEMENTS OF FORTRAN

4.1 Variable Names

A *name* is a symbolic link to a location in memory. A *variable* is a memory location whose value may be changed during execution. Names must:

- have between 1 and 31 alphanumeric characters (alphabet, digits and underscore);
- start with a letter.

It is possible – but unwise – to use a Fortran keyword or the name of a standard intrinsic function as a variable name. Tempting names that should be avoided in this respect include: COUNT, LEN, PRODUCT, RANGE, SCALE, SIZE, SUM and TINY.

The following are valid (if unlikely) variable names:

```
Manchester_United
AS_EASY_AS_123
STUDENT
```

The following are not:

```
ROMEO+JULIET      (+ is not allowed)
999Help           (starts with a number)
HELLO!            (! would be treated as a comment, not part of the variable name)
```

4.2 Data Types

In Fortran there are 5 *intrinsic* (i.e. built-in) data types:

```
integer
real
complex
character
logical
```

The first three are the *numeric* types. The last two are *non-numeric* types.

It is also possible to have *derived* types and *pointers*. Both of these are highly desirable in a modern programming language (they are very similar to features in the C programming language), but they are beyond the scope of this course.

Integer constants are whole numbers, without a decimal point, e.g.

```
100  +17  -444  0  666
```

They are stored exactly, but their range is limited: typically -2^{n-1} to $2^{n-1}-1$, where n is either 16 (for 2-byte integers) or 32 (for 4-byte integers – the default for our compiler). It is possible to change the default range using the `kind` type parameter (see later).

Real constants have a decimal point and may be entered as either

```
fixed point, e.g. 412.2
floating point, e.g. 4.122E+02
```

Real constants are stored in exponential form in memory, no matter how they are entered. They are accurate only to a finite machine precision (which, again, can be changed using the `kind` type parameter).

Complex constants consist of paired real numbers, corresponding to real and imaginary parts. e.g. $(2.0, 3.0)$ corresponds to $2 + 3i$.

Character constants consist of strings of characters enclosed by a pair of delimiters, which may be either single (') or double (") quotes; e.g.

```
'This is a string'
"School of Mechanical, Aerospace and Civil Engineering"
```

The delimiters themselves are not part of the string.

Logical constants may be either `.TRUE.` or `.FALSE.`

4.3 Declaration of Variables

Type Declarations

Variables should be *declared* (that is, have their data type defined and memory set aside for them) before any executable statements. This is achieved by a *type declaration statement* of the form, e.g.,

```
INTEGER NUM
REAL X
COMPLEX Z
LOGICAL ANSWER
```

More than one variable can be declared in each statement. e.g.

```
INTEGER I, J, K
```

Initialisation

If desired, variables can be *initialised* in their type-declaration statement. In this case a *double colon* (: :) separator must be used. Thus, the above examples might become:

```
INTEGER :: NUM = 20
REAL :: X = 0.05
COMPLEX :: Z = (0.0,1.0)
LOGICAL :: ANSWER = .TRUE.
```

Variables can also be initialised with a DATA statement; e.g.

```
DATA NUM, X, Z, ANSWER / 20, 0.05, (0.0,1.0), .TRUE. /
```

The DATA statement must be placed before any executable statements.

Attributes

Various *attributes* may be specified for variables in their type-declaration statements. One such is PARAMETER. A variable declared with this attribute may not have its value changed within the program unit. It is often used to emphasise key physical or mathematical constants; e.g.

```
REAL, PARAMETER :: PI = 3.14159
REAL, PARAMETER :: GRAVITY = 9.81
```

Other attributes will be encountered later and there is a list of them in the Appendix. Note that the double colon separator (: :) must be used when attributes are specified or variables are initialised – it is optional otherwise.

Precision and “Kind” (Optional)

By default, in the particular Fortran implementation in the University clusters a variable declared by, e.g.,

```
REAL X
```

will occupy 4 bytes of computer memory and will be inaccurate in the sixth significant figure. The accuracy can be increased by replacing the type statement by the often-used, but deprecated,

```
DOUBLE PRECISION X
```

with the floating-point variable now requiring twice as many bytes of memory.

Unfortunately, the number of bytes with which REAL and DOUBLE PRECISION floating-point numbers are stored is not set by the Fortran standard and varies between Fortran implementations. Similarly, whether an INTEGER is stored using 2, 4 or 8 bytes affects the largest number that can be represented exactly. On occasion these issues of accuracy and range may lead to different results from the same program run on different computers.

The *kind* concept will not be mentioned much in this short course, but it is valuable in ensuring true portability across platforms and one should be aware of its existence. If you wish true portability then you may wish to declare the *kind* of a variable type explicitly; e.g.

```

INTEGER, PARAMETER :: IKIND = SELECTED_INT_KIND(5)
INTEGER, PARAMETER :: RKIND = SELECTED_REAL_KIND(6,99)
INTEGER (KIND=IKIND) I
REAL (KIND=RKIND) R

```

In this example, the first two lines work out the kind type parameters needed to store integers of up to 5 digits (i.e. -99999 to 99999) and real numbers of accuracy at least 6 significant figures and covering a range -10^{99} to 10^{99} . These are assigned to parameter variables `IKIND` and `RKIND`, which can then be used to endow the integer `I` and the real `R` with the required range and precision.

To discover the default kinds for a particular Fortran implementation try

```
PRINT *, KIND( 1 ), KIND( 1.0 ), KIND( 1.0D0 )
```

The intrinsic function `KIND` returns the kind of its argument: in this example for default integer, real and double-precision data. (A quick test with the FTN95, F95 and G95 compilers produces “3 1 2”, “3 1 2” and “4 4 8”, indicating that the default kinds are not the same for different compilers even with the same computer and operating system.)

Historical Baggage – Implicit Typing.

Unless a variable was explicitly typed, older versions of Fortran implicitly assumed a type for a variable depending on the first letter of its name. Thus, if not explicitly declared, a variable whose name started with one of the letters I–O was assumed to be an integer; otherwise it was assumed to be real. (Hence the appalling Fortran joke: “GOD is REAL, unless declared INTEGER”!).

To allow older code to run, Fortran has to permit implicit typing. However, it is *very* bad programming practice (leading to major errors if you mis-type a variable: e.g. ANGEL instead of ANGLE) and it is highly advisable to:

- use a type declaration for all variables;
- put the `IMPLICIT NONE` statement at the start of all program units (this turns off implicit typing and the compiler will complain about any variable that you have forgotten to declare).

4.4 Numeric Operators and Expressions

A *numeric expression* is a formula combining constants, variables and functions using the *numeric intrinsic operators* given in the following table.

<i>operator</i>	<i>meaning</i>	<i>precedence (1 = highest)</i>
**	exponentiation (x^y)	1
*	multiplication (xy)	2
/	division (x/y)	2
+	addition ($x+y$) or unary plus ($+x$)	3
-	subtraction ($x-y$) or unary minus ($-x$)	3

An operator with two operands is called a *binary* operator. An operator with one operand is called a *unary* operator.

Precedence

Expressions are evaluated in order: highest precedence (exponentiation) first, then left to right. Brackets (), which have highest precedence of all, can be used to override this. e.g.

```

1 + 2 * 3           evaluates as 1 + (2 × 3)   or 7
10.0 / 2.0 * 5.0   evaluates as (10.0 / 2.0) × 5.0 or 25.0
5.0 * 2.0 ** 3     evaluates as 5.0 × (2.03)   or 40.0

```

Repeated exponentiation is the single exception to the left-to-right rule for equal precedence:

```
A ** B ** C           evaluates as ABC
```

Type Coercion

When a binary operator has operands of different type, the weaker (usually integer) type is *coerced* (i.e. converted) to the stronger (usually real) type and the result is of the stronger type. e.g.

$3 / 10.0 \rightarrow 3.0 / 10.0 \rightarrow 0.3$

*** WARNING *** The commonest source of difficulty is with *integer division*. If an integer is divided by an integer then the result must be an integer and is obtained by *truncation towards zero*. Thus, in the above example, if we had written $3/10$ (without a decimal point) the result would have been 0.

Integer division is fraught with dangers to the unwary. Be careful when mixing reals and integers in *mixed-mode* expressions. If you intend a constant to be a real number, *use a decimal point!*

Integer division can, however, be useful. For example,

$25 - 4 * (25 / 4)$

gives the remainder (here, 1) when 25 is divided by 4.

Type coercion also occurs in assignment. (“=” is formally an operator – albeit one with the lowest precedence of all, being performed after everything else is evaluated.) In this case, however, the conversion is to the type of the variable being assigned. Suppose I has been declared as an integer. Then it is only permitted to hold whole-number values and the statement

$I = -25.0 / 4.0$

will first evaluate the RHS (as -6.25) and then truncate it towards zero, assigning the value -6 to I.

4.5 Character Operators

There is only one character operator, *concatenation*, //; e.g.

'Man' // 'chester' gives 'Manchester'

4.6 Logical Operators and Expressions

A *logical expression* is either:

- a combination of numerical expressions and the *relational operators*
 - < less than
 - <= less than or equal
 - > greater than
 - >= greater than or equal
 - == equal
 - /= not equal
- a combination of other logical expressions, variables and the *logical operators* given below.

operator	meaning	precedence (1=highest)
.NOT.	logical negation (.TRUE. \rightarrow .FALSE. and vice-versa)	1
.AND.	logical intersection (both are .TRUE.)	2
.OR.	logical union (at least one is .TRUE.)	3
.EQV.	logical equivalence (both .TRUE. or both .FALSE.)	4
.NEQV.	logical non-equivalence (one is .TRUE. and the other .FALSE.)	4

As with numerical expressions, brackets can be used to override precedence.

A logical variable can be assigned to directly; e.g.

$L = .TRUE.$

or by using a logical expression; e.g.

$L = A > 0.0 .AND. C > 0.0$

Logical expressions are most widely encountered in decision making; e.g.

```
IF ( DISCRIMINANT < 0.0 ) PRINT *, 'Roots are complex'
```

The older forms `.LT.`, `.LE.`, `.GT.`, `.GE.`, `.EQ.`, `.NE.` may be used instead of `<`, `<=`, `>`, `>=`, `==`, `/=` if desired.

Character strings can also be compared, according to the *character-collating sequence* used by the compiler: this is often (but does not have to be), ASCII or EBCDIC. The Fortran standard requires that for all-upper-case, all-lower-case or all-numeric expressions, normal dictionary order is preserved. Thus, for example, both the logical expressions

```
'ABCD' < 'EF'  
'0123' < '3210'
```

are true, but

```
'DR' < 'APSLEY'
```

is false. However, upper case may or may not come before lower case in the character-collating sequence and letters may or may not come before numbers, so that mixed-case expressions or mixed alphabetic-numeric expressions should not be compared as they could conceivably give different answers on different platforms.

4.7 Line Discipline

The usual layout of statements is one-per-line, interspersed with blank lines for clarity. This is the recommended form in most instances. However,

- There may be more than one statement per line, separated by a *semicolon*; e.g.

```
A = 1; B = 10; C = 100
```

This is only recommended for simple initialisation of related variables.

- Each statement may run onto one or more *continuation lines* if there is an *ampersand* (&) at the end of the line to be continued. e.g.

```
RADIANS = DEGREES * PI &  
          / 180.0
```

is the same as the single-line statement

```
RADIANS = DEGREES * PI / 180.0
```

There may be up to 132 characters per line.

4.8 Miscellaneous Remarks

Pi

The constant π appears a lot in mathematical programming, e.g. when converting between degrees and radians. If a REAL variable PI is declared then its value can be set within the program:

```
PI = 3.14159
```

but it is neater to declare it as a PARAMETER in its type statement:

```
REAL, PARAMETER :: PI = 3.14159
```

Alternatively, a traditional method to obtain an accurate value is to invert the result $\tan(\pi/4) = 1$:

```
PI = 4.0 * ATAN( 1.0 )
```

This requires an expensive function evaluation, so should be done only once in a program.

Exponents

If an exponent (“power”) is coded as an integer (i.e. without a decimal point) it will be worked out by repeated multiplication; e.g.

```
A ** 3      will be worked out as    A * A * A
```

```
A ** (-3)   will be worked out as    1 / (A * A * A)
```

For non-integer powers (which includes whole-number powers if a decimal point is used) the result will be worked out by using the fact that

$$a^b = (e^{\ln a})^b = e^{b \ln a}$$

(Actually, the base may not be e , but the premise is the same.) e.g.

`A ** 3.0` will be worked out as something akin to $e^{3.0 \ln A}$.

However, *logarithms of negative numbers don't exist*, so the following Fortran statement is legitimate:

```
X = (-1) ** 2
```

but the next one isn't:

```
X = (-1) ** 2.0
```

The bottom line is that:

- if the exponent is genuinely a whole number, then don't use a decimal point – or, for small powers, simply write it explicitly as a repeated multiple: e.g. `A * A * A`
- take special care with odd roots of negative numbers; e.g. $(-1)^{1/3}$; you should work out the fractional power of the magnitude, then adjust the sign; e.g. write $(-8)^{1/3}$ as $-(8)^{1/3}$.

Remember: because of INTEGER arithmetic, the Fortran statement

```
X ** (1 / 3)
```

will actually evaluate as `X ** 0` (which is 1.0, and presumably not intended). To ensure REAL arithmetic, code the statement as

```
X ** (1.0 / 3.0)
```

A useful intrinsic function for setting the sign of an expression is

`SIGN(X, Y)` → absolute value of X times the sign of Y

5. REPETITION: DO AND DO WHILE

See Sample Programs – Week 2

5.1 Types of DO Loop

One advantage of computers is that they never get bored by repeating the same action many times.

If a block of code is to be performed repeatedly it is put inside a DO loop, the basic structure of which is:

```
DO . . .  
    repeated section  
END DO
```

(Indentation helps to clarify the logical structure of the code – it is easy to see which section is being repeated.)

There are two basic types of DO loops:

(a) *Deterministic* DO loops – the number of times the section is repeated is stated explicitly; e.g.,

```
DO I = 1, 10  
    repeated section  
END DO
```

This will perform the repeated section once for each value of the *counter* $I = 1, 2, 3, \dots, 10$. The value of I itself may or may not actually be used in the repeated section.

(b) *Non-deterministic* DO loops – the number of repetitions is not stated in advance. The enclosed section is repeated until some condition is or is not met. This may be done in two alternative ways. The first requires a logical reason for *stopping* looping, whilst the second requires a logical reason for *continuing* looping.

```
DO  
    ...  
    IF ( logical expression ) EXIT  
    ...  
END DO
```

or

```
DO WHILE ( logical expression )  
    repeated section  
END DO
```

5.2 Deterministic DO Loops

The general form of the DO statement in this case is:

```
DO variable = value1, value2 [ , value3 ]
```

Note that:

- the loop will execute for each value of the *variable* from *value1* to *value2* in steps of *value3*.

- *value3* may be negative or positive; if it is omitted (which is the usual case) then it is assumed to be 1;
- the counter *variable* must be of INTEGER type; (there could be round-off errors if using REAL variables);
- *value1*, *value2* and *value3* may be constants (e.g. 100) or expressions evaluating to integers (e.g. $6 * (2 + J)$)

The counter (L in the program below) may, as in this program, simply count the number of loops.

```
PROGRAM LINES
! Illustration of DO-loops
  IMPLICIT NONE
  INTEGER L                               ! a counter

  DO L = 1, 100                            ! start of repeated section
    PRINT *, 'I must not talk in class'
  END DO                                   ! end of repeated section

END PROGRAM LINES
```

Alternatively, the counter (I in the program below) may actually be used in the repeated section.

```
PROGRAM DOLOOPS
  IMPLICIT NONE
  INTEGER I

  DO I = 1, 20
    PRINT *, I, I * I
  END DO

END PROGRAM DOLOOPS
```

Observe the effect of changing the DO statement to, for example,

```
DO I = 10, 20, 3
or
DO I = 20, -20, -5
```

5.3 Non-Deterministic DO Loops

The

```
IF ( ... ) EXIT
```

form continues until some logical expression evaluates as `.TRUE.`. Then it jumps out of the loop and continues with the code after the loop. In this form a `.TRUE.` result tells you when to *stop* looping.

The

```
DO WHILE ( ... )
```

form continues until some logical expression evaluates as `.FALSE.`. Then it stops looping and continues with the code after the loop. In this form a `.TRUE.` result tells you when to *continue* looping.

Most problems involving non-deterministic loops can be written in either form, although some programmers express a preference for the latter because it makes clear in an easily identified place (the top of the loop) the criterion for looping.

Non-deterministic DO-loops are particularly good for

- summing power series (looping stops when the absolute value of a term is less than some given tolerance);
- single-point iteration (looping stops when the change is less than a given tolerance).

As an example of the latter consider the following code for solving the Colebrook-White equation for the friction factor λ in flow through a pipe:

$$\frac{1}{\sqrt{\lambda}} = -2.0 \log_{10} \left(\frac{k_s}{3.7D} + \frac{2.51}{\text{Re} \sqrt{\lambda}} \right)$$

The user inputs values of the relative roughness (k_s/D) and Reynolds number Re. For simplicity, the program actually iterates for $x = 1/\sqrt{\lambda}$:

$$x = -2.0 \log_{10} \left(\frac{k_s}{3.7D} + \frac{2.51}{\text{Re}} x \right)$$

Note that:

- x must have a starting value (here it is set to 1);
- there must be a criterion for continuing or stopping iteration (here: looping/iteration continues until successive values differ by less than some tolerance, say 10^{-5});
- in practice, this calculation would probably be part of a much bigger pipe-network calculation and would be better coded as a *function* (see Section 9) rather than a *main program*.

```

PROGRAM FRICTION
  IMPLICIT NONE
  REAL KSD                                ! relative roughness (ks/D)
  REAL RE                                  ! Reynolds number
  REAL X                                   ! 1/SQRT(lambda)
  REAL XOLD                                ! previous value of X
  REAL, PARAMETER :: TOLERANCE = 1.0E-5  ! convergence tolerance

  PRINT *, 'Input ks/D and Re'           ! request values
  READ *, KSD, RE

  X = 1.0                                 ! initial guess
  XOLD = X + 1.0                          ! anything different from X

  DO WHILE ( ABS( X - XOLD ) > TOLERANCE )
    XOLD = X                               ! store previous
    X = -2.0 * LOG10( KSD / 3.7 + 2.51 * X / RE ) ! new value
  END DO

  PRINT *, 'Friction factor = ', 1.0 / ( X * X ) ! output lambda

END PROGRAM FRICTION

```

Exercise: re-code the DO loop repeat criterion in the IF (...) EXIT form.

5.4 Nested DO Loops

DO loops can be *nested* (i.e. one inside another). Indentation is definitely recommended here to clarify the loop structure. A rather unspectacular example is given below.

```

PROGRAM NESTED
! Illustration of nested DO-loops
  IMPLICIT NONE
  INTEGER I, J                               ! loop counters

  DO I = 1, 5                                 ! start of outer loop
    DO J = 1, 3                               ! start of inner loop
      PRINT *, 'I, J = ', I, J
    END DO                                    ! end of inner loop
  END DO                                      ! end of outer loop

END PROGRAM NESTED

```

5.5 Non-Integer Steps

The DO loop counter must be an INTEGER (to avoid round-off errors). To increment x in a non-integer sequence, e.g

0.5, 0.8, 1.1, ...

you should work out successive values in terms of a separate integral counter by specifying:

- an initial value (x_0);
- a step size (Δx)
- the number of values to be output (N_x).

The successive values are:

$x_0, x_0 + \Delta x, x_0 + 2\Delta x, \dots, x_0 + (N_x - 1)\Delta x.$

The i^{th} value is

$x_0 + (i - 1)\Delta x$ for $i = 1, \dots, N_x.$

```
PROGRAM XLOOP
! Illustration of non-integer values
IMPLICIT NONE
REAL X           ! value to be output
REAL X0          ! initial value of X
REAL DX          ! increment in X
INTEGER NX       ! number of values
INTEGER I        ! loop counter

PRINT *, 'Input X0, DX, NX' ! request values
READ *, X0, DX, NX

DO I = 1, NX     ! start of repeated section
  X = X0 + ( I - 1 ) * DX ! value to be output
  PRINT *, X
END DO          ! end of repeated section

END PROGRAM XLOOP
```

If one only uses the variable X once for each of its values (as in the example above) there is no need to define it as a separate variable, and one could simply combine the lines

```
X = X0 + ( I - 1 ) * DX
PRINT *, X
```

as

```
PRINT *, X0 + ( I - 1 ) * DX
```

There is then no need for a separate variable X.

6. DECISION-MAKING: IF AND CASE

See Sample Programs – Week 2

Often a computer is called upon to perform one set of actions if some condition is met, and (optionally) some other set if it is not. This *branching* or *conditional* action can be achieved by the use of IF or CASE constructs. A very simple use of IF . . . ELSE was given in the quadratic-equation program of Section 3.

6.1 The IF Construct

There are several forms of IF construct.

(i) Single statement.

```
IF ( logical expression ) statement
```

(ii) Single block of statements.

```
IF ( logical expression ) THEN
```

```
    things to be done if true
```

```
END IF
```

(iii) Alternative actions.

```
IF ( logical expression ) THEN
```

```
    things to be done if true
```

```
ELSE
```

```
    things to be done if false
```

```
END IF
```

(iv) Several alternatives (there may be several ELSE IFs, and there may or may not be an ELSE).

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

```
    .....
```

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

```
    .....
```

```
[ ELSE
```

```
    .....
```

```
]
```

```
END IF
```

As with DO loops, IF constructs can be nested; (this is where indentation is very helpful).

6.2 The CASE Construct

The CASE construct is a convenient (and sometimes more readable and/or efficient) alternative to an IF ... ELSE IF ... ELSE construct. It allows different actions to be performed depending on the *set* of outcomes (*selector*) of a particular expression.

The general form is:

```
SELECT CASE ( expression )
  CASE ( selector-1 )
    block-1
  CASE ( selector-2 )
    block-2
  [ CASE DEFAULT
    default block
  ]
END SELECT
```

expression is an integer, character or logical expression. It is often just a simple variable.

selector-n is a set of values that *expression* might take.

block-n is the set of statements to be executed if *expression* lies in *selector-n*.

CASE DEFAULT is used if *expression* does not lie in any other category. It is optional.

Selectors are lists of non-overlapping integer or character outcomes, separated by commas. Outcomes can be individual values (e.g. 3, 4, 5, 6) or ranges (e.g. 3:6). These are illustrated below and in the week's examples.

Example. What type of key am I pressing?

```
PROGRAM KEYPRESS
  IMPLICIT NONE

  CHARACTER LETTER

  PRINT *, 'Press a key'
  READ *, LETTER

  SELECT CASE ( LETTER )

    CASE ( 'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U' )
      PRINT *, 'Vowel'

    CASE ( 'b':'d', 'f':'h', 'j':'n', 'p':'t', 'v':'z', &
           'B':'D', 'F':'H', 'J':'N', 'P':'T', 'V':'Z' )
      PRINT *, 'Consonant'

    CASE ( '0':'9' )
      PRINT *, 'Number'

    CASE DEFAULT
      PRINT *, 'Something else'

  END SELECT

END PROGRAM KEYPRESS
```

7. ARRAYS

See Sample Programs – Week 2

In geometry it is common to denote coordinates by x_1, x_2, x_3 or $\{x_i\}$. The elements of matrices are written as $a_{11}, a_{12}, \dots, a_{mn}$ or $\{a_{ij}\}$. These are examples of *subscripted variables* or *arrays*.

It is common and convenient to denote the whole array by its unsubscripted name; e.g. $\mathbf{x} \equiv \{x_i\}$, $\mathbf{a} \equiv \{a_{ij}\}$. The presence of subscripted variables is important in any programming language. The ability to refer to an array as a whole, without subscripts, is an element of Fortran 90/95 which makes it particularly useful in engineering.

When referring to an individual *element* of an array, the subscripts are enclosed in parentheses; e.g. $X(1)$, $A(1, 2)$, etc..

7.1 One-Dimensional Arrays (Vectors)

Example. Consider the following program to fit a straight line to the set of points (x_1, y_1) , (x_2, y_2) , \dots , (x_N, y_N) and then print them out, together with the best-fit straight line. The data file is assumed to be of the form shown right and the best-fit straight line is $y = mx + c$ where

$$m = \frac{\frac{\sum xy}{N} - \bar{x} \bar{y}}{\frac{\sum x^2}{N} - \bar{x}^2}, \quad c = \bar{y} - m\bar{x} \quad \text{where} \quad \bar{x} = \frac{\sum x}{N}, \quad \bar{y} = \frac{\sum y}{N}$$

N	
x_1	y_1
x_2	y_2
\dots	
x_N	y_N

```

PROGRAM LINE_1
  IMPLICIT NONE
  INTEGER N                                ! number of points
  INTEGER I                                ! a counter
  REAL X(100), Y(100)                      ! arrays to hold the points
  REAL SUMX, SUMY, SUMXY, SUMXX           ! various intermediate sums
  REAL M, C                                ! line slope and intercept
  REAL XBAR, YBAR                          ! mean x and y

  SUMX = 0.0; SUMY = 0.0; SUMXY = 0.0; SUMXX = 0.0      ! initialise sums

  OPEN( 10, FILE = 'pts.dat' )              ! open data file; attach to unit 10
  READ( 10, * ) N                            ! read number of points

  ! Read rest of marks, one per line, and add to sums
  DO I = 1, N
    READ( 10, * ) X(I), Y(I)
    SUMX = SUMX + X(I)
    SUMY = SUMY + Y(I)
    SUMXY = SUMXY + X(I) * Y(I)
    SUMXX = SUMXX + X(I) ** 2
  END DO
  CLOSE( 10 )                                ! finished with data file

  ! Calculate best-fit straight line
  XBAR = SUMX / N
  YBAR = SUMY / N
  M = ( SUMXY / N - XBAR * YBAR ) / ( SUMXX / N - XBAR ** 2 )
  C = YBAR - M * XBAR

  PRINT *, 'Slope = ', M
  PRINT *, 'Intercept = ', C
  PRINT '( 3( 1X, A10 ) )', 'x', 'y', 'mx+c'
  DO I = 1, N
    PRINT '( 3( 1X, 1PE10.3 ) )', X(I), Y(I), M * X(I) + C
  END DO

END PROGRAM LINE_1

```

Several features of arrays can be illustrated by this example.

7.2 Array Declaration

Like any other variables, arrays need to be declared at the start of a program unit and memory space assigned to them. However, unlike *scalar* variables, array declarations require both a *type* (integer, real, complex, character, logical, ...) and a *size* (i.e. number of elements).

In this case the two one-dimensional arrays X and Y can be declared as of real type with 100 elements by the type-declaration statement

```
REAL X(100), Y(100)
```

or using the DIMENSION attribute:

```
REAL, DIMENSION(100) :: X, Y
```

or by a separate DIMENSION statement:

```
REAL X, Y  
DIMENSION X(100), Y(100)
```

By default, the first element of an array has subscript 1. It is possible to make the array start from subscript 0 (or any other integer) by declaring the lower array bound as well. For example, to start at 0 instead of 1:

```
REAL X(0:100)
```

Warning: in the C programming language the default lowest subscript is 0.

7.3 Dynamic Arrays

An obvious problem arises. What if the number of points N is greater than the declared size of the array (here, 100)? Well, different compilers will do different and unpredictable things – most resulting in crashes.

One solution (which used to be required in earlier versions of Fortran) was to check for adequate space, prompting the user to recompile if necessary with a larger array size:

```
READ ( 10, * ) N  
IF ( N > 100 ) THEN  
    PRINT *, 'Sorry, N > 100. Please recompile with larger array'  
    STOP  
END IF
```

It is probably better to keep out of the way of administrators if they encounter this error message.

A far better solution is to use *dynamic memory allocation*; that is, the array size is determined (and computer memory allocated) at run-time, not in advance during compilation. To do this one must use *allocatable* arrays as follows.

(i) In the declaration statement, use the ALLOCATABLE attribute; e.g.

```
REAL, ALLOCATABLE :: X(:), Y(:)
```

Note that the *shape*, but not *size*, is indicated *at compile-time* by a single colon (:).

(ii) Once the size of the arrays has been identified *at run-time*, allocate them the required amount of memory:

```
READ ( 10, * ) N  
ALLOCATE ( X(N), Y(N) )
```

(iii) When the arrays are no longer needed, recover memory by de-allocating them:

```
DEALLOCATE ( X, Y )
```

7.4 Array Input/Output and Implied DO Loops

In the example, the lines

```
DO I = 1, N  
    READ ( 10, * ) X(I), Y(I)  
    ...  
END DO
```

mean that at most one pair of points can be input per line. With the single statement

```
READ ( 10, * ) ( X(I), Y(I), I = 1, N )
```

the program will simply read the first N data pairs (separated by spaces, commas or line breaks) that it encounters. Since all the points are read in one go, they no longer need to be on separate lines of the input file. The loop is implied without an actual DO statement, so this is called (reasonably enough!) an *implied DO loop*.

7.5 Array-handling Functions

Certain intrinsic functions are built into the language to facilitate array handling. For example, the one-by-one summation can be replaced by the single statement

```
SUMX = SUM( X )
```

This uses the intrinsic function SUM, which adds together all elements of its array argument. Other array-handling functions are listed in Appendix A4 and in the recommended textbooks.

7.6 Element-by-Element Operations

Sometimes we want to do the same thing to every element of an array. In the above example, for each mark we form the square of that mark and add to a sum. The *array expression*

```
X * X
```

is a new array with elements $\{x_i^2\}$. `SUM(X * X)` therefore produces $\sum x_i^2$.

Using many of these array features a shorter version of the program is given below. Note that use of the intrinsic function SUM obviates the need for extra variables to hold intermediate sums and there is a one-line implied DO loop for both input and output.

```
PROGRAM LINE_2
  IMPLICIT NONE
  INTEGER N                                ! number of points
  INTEGER I                                ! a counter
  REAL, ALLOCATABLE :: X(:), Y(:)         ! arrays to hold the points
  REAL M, C                                 ! line slope and intercept
  REAL XBAR, YBAR                           ! mean x and y

  OPEN( 10, FILE = 'pts.dat' )             ! open data file; attach to unit 10
  READ( 10, * ) N                           ! read number of points
  ALLOCATE( X(N), Y(N) )                   ! allocate memory to X and Y
  READ( 10, * ) ( X(I), Y(I), I = 1, N )   ! read rest of marks
  CLOSE( 10 )                               ! finished with data file

  ! Calculate best-fit straight line
  XBAR = SUM( X ) / N
  YBAR = SUM( Y ) / N
  M = ( SUM( X * Y ) / N - XBAR * YBAR ) / ( SUM( X * X ) / N - XBAR ** 2 )
  C = YBAR - M * XBAR

  PRINT *, 'Slope = ', M
  PRINT *, 'Intercept = ', C
  PRINT '( 3( 1X, A10 ) )', 'x', 'y', 'mx+c'
  PRINT '( 3( 1X, 1PE10.3 ) )', ( X(I), Y(I), M * X(I) + C, I = 1, N )

  DEALLOCATE( X, Y )                       ! retrieve memory space
END PROGRAM LINE_2
```

7.7 Matrices and Higher-Dimension Arrays

An $m \times n$ array of numbers of the form

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & & a_{mn} \end{pmatrix}$$

is called a *matrix*. The typical element is denoted a_{ij} . It has two subscripts.

Fortran allows matrices (two-dimensional arrays) and, in fact, arrays of up to 7 dimensions. (However, entities of the form $a_{ijklmno}$ have never found much application in engineering!)

In Fortran the declaration and use of a REAL 3×3 matrix might look like

```
REAL A(3,3)
A(1,1) = 1.0;   A(1,2) = 2.0;   A(1,3) = 3.0
A(2,1) = 4.0
etc.
```

Other methods of initialisation will be discussed below.

Matrix Multiplication

Suppose A, B and C are 3×3 matrices declared by

```
REAL, DIMENSION(3,3) :: A, B, C
```

The statement

```
C = A * B
```

does *element-by-element* multiplication; i.e. each element of C is the product of the corresponding elements in A and B.

To do “proper” matrix multiplication use the standard MATMUL function:

```
C = MATMUL( A, B )
```

Obviously matrix multiplication is not restricted to 3×3 matrices. However, for matrix multiplication to be legitimate the matrices must be *conformable*; i.e. the number of columns of A must equal the number of rows of B.

7.8 Array Initialisation

Sometimes it is necessary to initialise all elements of an array. This can be done by separate statements; e.g.

```
A(1) = 1.0;   A(2) = 20.5;   A(3) = 10.0;   A(4) = 0.0;   A(5) = 0.0
```

It can also be done with a DATA statement:

```
DATA A / 1.0, 20.5, 10.0, 0.0, 0.0 /
```

DATA statements can be used to initialise multi-dimensional arrays. However, the storage order of elements is important. In Fortran, *column-major* storage is used; i.e. the first subscript varies fastest. For example, the storage order of a 3×3 matrix is

```
A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2), A(1,3), A(2,3), A(3,3)
```

Warning: this is the opposite convention to the C programming language.

7.9 Array Assignment and Array Expressions

Arrays are used where large numbers of data elements are to be treated in similar fashion. Fortran 90/95 now allows a “syntactic shorthand” to be used whereby, if the array name is used in a numeric expression without subscripts, then the operation is assumed to be performed on *every element* of an array. This is far more concise than older versions of Fortran, where it was necessary to use DO-loops.

For example, suppose that arrays X, Y and Z are declared with, say, 10 elements:

```
REAL, DIMENSION(10) :: X, Y, Z
```

Assignment

```
X = 5.0
```

sets every element of X to the value 5.0.

Array Expressions

```
Y = -3 * X
```

Sets y_i to $-3x_i$ for each element of the respective arrays.

```
Y = X + 3
```

Although 3 is only a scalar, y_i is set to $x_i + 3$ for each element of the arrays.

```
Z = X * Y
```

Sets z_i to $x_i y_i$ for each element of the respective arrays. Remember: this is “element-by-element” multiplication.

Array Arguments to Intrinsic Functions

```
Y = SIN( X )
```

Sets y_i to $\sin(x_i)$ for each element of the respective arrays.

7.10 The WHERE Construct

WHERE is simply an IF construct applied to every element of an array. For example, to turn every non-zero element of an array A into its reciprocal, one could write

```
WHERE ( A /= 0.0 )  
  A = 1.0 / A  
END WHERE
```

Note that the individual elements of A are never mentioned. {WHERE, ELSE, ELSE WHERE, END WHERE} can be used whenever one wants to use a corresponding {IF, ELSE, ELSE IF, END IF} for each element of an array.

8. CHARACTER HANDLING

See Sample Programs – Week 3

Fortran (FORMula TRANslation) was originally developed for scientific and engineering calculations, not word-processing. However, modern versions now have extensive character-handling capabilities.

8.1 Character Constants and Variables

A *character constant* (or *string*) is a series of characters enclosed in delimiters, which may be either single (') or double (") quotes; e.g.

```
'This is a string' or "This is a string"
```

The delimiters themselves are not part of the string.

Delimiters of the opposite type can be used within a string with impunity; e.g.

```
PRINT *, "This isn't a problem"
```

However, if the bounding delimiter is to be included in the string then it must be doubled up; e.g.

```
PRINT *, 'This isn''t a problem.'
```

Character variables must have their *length* – i.e. number of characters – declared in order to set aside memory. Any of the following will declare a character variable WORD of length 10:

```
CHARACTER (LEN=10) WORD
```

```
CHARACTER (10) WORD
```

```
CHARACTER WORD*10
```

(The first is my personal preference).

To save counting characters, an assumed length (indicated by LEN=* or, simply, *) may be used for character variables with the PARAMETER attribute; i.e. those whose value is fixed. e.g.

```
CHARACTER (LEN=*), PARAMETER :: UNIVERSITY = 'MANCHESTER'
```

If LEN is not specified for a character variable then it defaults to 1; e.g.

```
CHARACTER LETTER
```

Character arrays are simply subscripted character variables. Their declaration requires a dimension statement in addition to length; e.g.

```
CHARACTER (LEN=3), DIMENSION(12) :: MONTHS
```

or, equivalently,

```
CHARACTER (LEN=3) MONTHS(12)
```

This array might then be initialised by, for example,

```
DATA MONTHS / 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', &  
              'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec' /
```

8.2 Character Assignment

When character variables are assigned they are filled from the left and padded with blanks if necessary. For example, if UNIVERSITY is a character variable of length 7 then

```
UNIVERSITY = 'UMIST'           fills UNIVERSITY with 'UMIST  '
```

```
UNIVERSITY = 'Manchester'     fills UNIVERSITY with 'Manches'
```

8.3 Character Operators

The only character operator is // (*concatenation*) which simply sticks two strings together; e.g.

```
'Man' // 'chester' → 'Manchester'
```

8.4 Character Substrings

Character substrings may be extended in a similar fashion to sub-arrays; (in a sense, a character string is an array – a vector of single characters). e.g. if CITY = 'Manchester' then

```
CITY(2:5)='anch'  
CITY(:3)='Man'  
CITY(7:)= 'ster'
```

8.5 Comparing and Ordering

Each computer system has a *character-collating sequence* that specifies the intrinsic ordering of the character set. Two of the most common are ASCII (shown below) and EBCDIC. 'Less than' (<) and 'greater than' (>) strictly refer to the position of the characters in this collating sequence.

	0	1	2	3	4	5	6	7	8	9
30			space	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~	del		

The ASCII character set. Characters 0-31 are control characters like [TAB] or [ESC] and are not shown.

The Fortran standard requires that upper-case letters A-Z and lower-case letters a-z are separately in alphabetical order, that the numerals 0-9 are in numerical order, and that a blank space comes before both. It does not, however, specify whether numbers come before or after letters in the collating sequence, or lower case comes before or after upper case. Provided there is consistent case, strings can be compared on the basis of dictionary order, but the standard gives no guidance when comparing letters with numerals or upper with lower case.

Example. The following logical expressions are both true (sadly):

```
'David Cameron' > 'David Apsley'  
'First year' < 'Second year'
```

Example.

```
100 < 20          gives .FALSE. as a numeric comparison  
'100' < '20'     gives .TRUE. as a string comparison (comparison based on the first character).
```

8.6 Intrinsic Subprograms With Character Arguments

The more common character-handling routines are given in Appendix A4. A full set is given in Hahn (1994).

Position in the Collating Sequence

CHAR(I) character in position I of the system collating sequence;
ICHAR(C) position of character C in the system collating sequence.

The system may or may not use ASCII as a collating system, but the following routines are always available:

ACHAR(I) character in position I of the ASCII collating sequence;
IACHAR(C) position of character C in the ASCII collating sequence.

The collating sequence may be used, for example, to sort names into alphabetical order or convert between upper and lower case, as in the following example.

Example. Since the separation of 'b' and 'B', 'c' and 'C' etc. in the collating sequence is the same as that between 'a' and 'A', the following subroutine may be used successively for each character to convert lower to upper case. If LETTER has lower case it will:

- convert to its number using ICHAR()
- add the numerical difference between upper and lower case: ICHAR('A') - ICHAR('a')
- convert back to a character using CHAR()

```
SUBROUTINE UC( LETTER )
  IMPLICIT NONE
  CHARACTER (LEN=1) LETTER

  IF ( LETTER >= 'a' .AND. LETTER <= 'z' ) THEN
    LETTER = CHAR( ICHAR( LETTER ) + ICHAR( 'A' ) - ICHAR( 'a' ) )
  END IF

END SUBROUTINE UC
```

Length of String

LEN(STRING)	declared length of STRING, even if it contains trailing blanks;
TRIM(STRING)	same as STRING but without any trailing blanks;
LEN_TRIM(STRING)	length of STRING with any trailing blanks removed.

Justification

ADJUSTL(STRING)	left-justified STRING
ADJUSTR(STRING)	right-justified STRING

Finding Text Within Strings

INDEX(STRING, SUBSTRING)	position of first (i.e. leftmost) occurrence of SUBSTRING in STRING
SCAN(STRING, SET)	position of first occurrence of <i>any</i> character from SET in STRING
VERIFY(STRING, SET)	position of first character in STRING that is <i>not</i> in SET

Each of these functions returns 0 if no such position is found.

To search for the *last* (i.e. rightmost) rather than first occurrence, add a third argument `.TRUE.`, e.g.:

```
INDEX( STRING, SUBSTRING, .TRUE. )
```

returns the position of the *last* occurrence of SUBSTRING in STRING.

9. FUNCTIONS AND SUBROUTINES

See Sample Programs – Week 3

All major computing languages allow complex and/or repetitive programs to be broken down into simpler procedures, each carrying out particular well-defined tasks, often with different values of their *arguments*. In Fortran these subprograms are called *subroutines* and *functions*. Examples of the action carried out by a single subprogram might be:

- calculate the distance $r = \sqrt{x^2 + y^2}$ of a point (x,y) from the origin;
- calculate $n! = n(n-1)...2.1$ for a positive integer n

As these are likely to be needed several times, it is appropriate to code them – just once – as a subprogram.

9.1 Intrinsic Subprograms

Certain subprograms – *intrinsic* subprograms – are defined by the Fortran standard and must be provided by an implementation's standard libraries. For example, the statement

```
Y = X * SQRT( X )
```

invokes an *intrinsic function* SQRT, with *argument* X, and *returns* a value (in this case, the square root of its argument) which is then employed to evaluate the numeric expression.

Useful mathematical intrinsic functions are listed in Appendix A4. The complete set required by the standard is given in Hahn (1994). Particular Fortran implementations may supply additional routines; for example, FTN95 includes many plotting routines and an interface (ClearWin+) to the Windows operating system.

9.2 Program Units

There are four types of *program unit*:

main programs
subroutines
functions
modules

Each source file may contain one or more program units and is compiled separately. (This is why one requires a link stage after compilation.) The advantage of separating subprograms between source files is that other programs can make use of the same routines.

Main Programs

Every Fortran program must contain exactly one *main program* which should start with a PROGRAM statement. This may invoke functions or subroutines which may, in turn, invoke other subprograms.

Subroutines

A subroutine is invoked by

```
CALL subroutine-name ( argument list )
```

The subroutine carries out some action according to the value of the arguments. It may or may not change the values of these arguments.

Functions

A function is invoked simply by using its name (and argument list) in a numeric expression; e.g.

```
DISTANCE = RADIUS( X, Y )
```

Within the function's source code its name (without arguments) is treated as a variable and should be assigned

a value, which is the value of the function on exit – see the example below. A function should be used when a single (usually numerical, but occasionally character or logical) value is to be returned. It is permissible, but poor practice, for a function to change its arguments – a better vehicle in that case would be a subroutine.

Modules (see Section 11)

Functions and subroutines may be *internal* (i.e. CONTAINED within and only accessible to one particular program unit) or *external* (and accessible to all). In this course we focus on the latter. Related internal routines are better gathered together in special program units called *modules*. Their contents are then made available collectively to other program units by the initial statement

```
USE module-name
```

The basic forms of main program, subroutines and functions are very similar and are given below. As usual, [] denotes something optional but, in these cases, it is strongly recommended.

Main program	Subroutines	Functions
<pre>[PROGRAM [name]] USE statements [IMPLICIT NONE] type declarations executable statements END [PROGRAM [name]]</pre>	<pre>SUBROUTINE name (argument-list) USE statements [IMPLICIT NONE] type declarations executable statements END [SUBROUTINE [name]]</pre>	<pre>[type] FUNCTION name (argument-list) USE statements [IMPLICIT NONE] type declarations executable statements END [FUNCTION [name]]</pre>

The first line is called the *subprogram statement* and defines the type of program unit, its name and its arguments. FUNCTION subprograms must also have a *type*. This may be declared in the subprogram statement or in a separate type declaration within the routine itself.

Subprograms pass control back to the calling program when they reach the END statement. Sometimes it is required to pass control back before this. This is effected by the RETURN statement. A similar early death can be effected in a main program by a STOP statement.

Many actions can be coded as either a function or a subroutine. For example, consider a program which calculates distance from the origin, $r = (x^2 + y^2)^{1/2}$:

(Using a function)	(Using a subroutine)
<pre>PROGRAM EXAMPLE IMPLICIT NONE REAL X, Y REAL, EXTERNAL :: RADIUS PRINT *, 'Input X, Y' READ *, X, Y PRINT *, 'Distance = ', RADIUS(X, Y) END PROGRAM EXAMPLE !===== REAL FUNCTION RADIUS(A, B) IMPLICIT NONE REAL A, B RADIUS = SQRT(A ** 2 + B ** 2) END FUNCTION RADIUS</pre>	<pre>PROGRAM EXAMPLE IMPLICIT NONE REAL X, Y REAL RADIUS EXTERNAL DISTANCE PRINT *, 'Input X, Y' READ *, X, Y CALL DISTANCE(X, Y, RADIUS) PRINT *, 'Distance = ', RADIUS END PROGRAM EXAMPLE !===== SUBROUTINE DISTANCE(A, B, R) IMPLICIT NONE REAL A, B, R R = SQRT(A ** 2 + B ** 2) END SUBROUTINE DISTANCE</pre>

Note that, in the first example, the calling program must declare the type of the function RADIUS amongst its other type declarations. It is optional, but good practice, to identify external functions or subroutines by using either an EXTERNAL attribute in the type statement (as in the first example) or a separate EXTERNAL statement (as in the second example). This makes clear what external routines are being used and ensures that if the Fortran implementation supplied an intrinsic routine of the same name then the external routine would override it.

Note that all variables in the functions or subroutines above have *scope* the program unit in which they are declared; that is, they have no connection with any variables of the same name in any other program unit.

9.3 Subprogram Arguments

The arguments in the subprogram statement are called *dummy arguments*: they exist only for the purpose of defining this subprogram and have no connection to other variables of the same name in other program units. The arguments used when the subprogram is actually invoked are called the *actual arguments*. They may be variables (e.g. X, Y), constants (e.g. 1.0, 2.0) or expressions (e.g. 3.0 + X, or 2.0 / Y), but they must be of the same type and number as the dummy arguments. For example, the RADIUS function above could not be invoked as RADIUS(X) (too few arguments) or as RADIUS(1, 2) (arguments of the wrong type).

(You may wonder how it is, then, that many intrinsic subprograms can be invoked with different types of argument. For example, in the statement

```
Y = EXP( X )
```

X may be real or complex, scalar or array. This is achieved by a useful, but highly-advanced, process known as *overloading*, which is way beyond the scope of this course.)

Passing by Name/Passing by Reference

In Fortran, if the actual arguments are variables, they are passed *by reference*, and their values will change if the values of the dummy arguments change in the subprogram. If, however, the actual arguments are either constants or expressions, then the arguments are passed *by value*; i.e. the values are copied into the subprogram's dummy arguments.

Warning: in C or C++ all arguments are passed by value – a feature that necessitates the use of *pointers* in order to change their values when returning to the original program.

Declaration of Intent

Because input variables passed as arguments may be changed unwittingly if the dummy arguments change within a subprogram, or, conversely, because a particular argument is intended as output and so must be assigned to a variable (not a constant or expression), it is good practice to declare whether dummy arguments are intended as input or output by using the INTENT attribute. e.g. in the above example:

```
SUBROUTINE DISTANCE( A, B, R )  
  REAL, INTENT( IN ) :: A, B  
  REAL, INTENT( OUT ) :: R
```

This signifies that dummy arguments A and B must not be changed within the subroutine and that the third actual argument must be a variable. There is also an INTENT(INOUT) attribute.

9.4 The SAVE Attribute

By default, variables declared within a subprogram do not retain their values between successive calls to the same subprogram. This behaviour can be overridden by the SAVE attribute; e.g.

```
REAL, SAVE :: X
```

which will store the value of X for the next time the routine is used.

9.5 Array Arguments

Arrays can be passed as arguments in much the same way as scalars, except that the subprogram must know the dimensions of the array. This can be achieved in a number of ways, the most common being:

- Fixed array size – usually for smaller arrays such as coordinate vectors; e.g.

```
SUBROUTINE GEOMETRY( X )  
  REAL X(3)
```

- Pass the array size as an argument; e.g.

```
SUBROUTINE GEOMETRY( NDIM, X )  
  REAL X(NDIM)
```

To avoid errors, array dummy arguments should have the same dimensions and shape as the actual arguments. Dummy arguments that are arrays must not have the `ALLOCATABLE` attribute. Their size must already have been declared or allocated in the invoking program unit.

9.6 Character Arguments

Dummy arguments of character type behave in a similar manner to arrays – their length must be made known to the subprogram. However, a character dummy argument may always be declared with assumed length (determined by the length of the actual argument); e.g.

```
CALL EXAMPLE( 'David' )  
...  
SUBROUTINE EXAMPLE( PERSON )  
  CHARACTER (LEN=*) PERSON
```

There are a large number of intrinsic character-handling routines (see the recommended textbooks). Some of the more useful ones are given in Appendix A4.

10. ADVANCED INPUT/OUTPUT

See Sample Programs – Week 4

Hitherto we have used simple *list-directed* input/output (i/o) with the *standard input/output devices* (keyboard and screen):

```
READ *, list
PRINT *, list
```

This section describes how to:

- use *formatted* output to control the layout of results;
- read from and write to *files*;
- use additional *specifiers* to provide advanced i/o control.

10.1 READ and WRITE

General list-directed i/o is performed by the statements

```
READ( unit, format ) list
WRITE( unit, format ) list
```

unit can be one of:

- an asterisk *, meaning the standard i/o device (usually the keyboard/screen);
- a *unit number* in the range 1 to 99 which has been associated with an *external file* (see below);
- a character variable (*internal file*): this is the simplest way of interconverting numbers and strings.

format can be one of:

- an asterisk *, meaning list-directed i/o;
- a *label* associated with a FORMAT statement containing a format specification;
- a character constant or expression evaluating to a format specification.

list is a set of variables or expressions to be input or output.

In terms of the simpler i/o statements used before:

```
READ( *, * ) is equivalent to READ *
WRITE( *, * ) is equivalent to PRINT *
```

10.2 Input/Output With Files

Before an external file can be read from or written to, it must be associated with a *unit number* by an OPEN statement. e.g. to associate the external file `input.dat` with the unit number 10:

```
OPEN( 10, FILE = 'input.dat' )
```

One can then read from the file using

```
READ( 10, ... ) ...
```

or write to the file using

```
WRITE( 10, ... ) ...
```

Although units are automatically disconnected at program end it is good practice (and it frees the unit number for re-use) if it is explicitly closed when no longer needed. For the above example, this means:

```
CLOSE( 10 )
```

In general the unit number (10 in the above example) may be any number in the range 1-99. Historically, however, 5 and 6 have been preconnected to the standard input and standard output devices, respectively.

The example above shows OPEN used to attach a file for *sequential* (i.e. beginning-to-end), *formatted* (i.e. human-readable) access. This is the default and is all we shall have time to cover in this course. However, Fortran can be far more flexible – see for example the recommended textbooks.

10.3 Formatted WRITE

In the output statement

```
WRITE( unit, format ) list
```

list is a comma-separated set of constants or variables to be output, *unit* indicates where the output is to go, whilst *format* indicates the way in which the output is to be set out. If *format* is an asterisk * then the computer will choose how to set it out. However, if you wish to display output in a particular way, for example in neat columns, then you must specify the format more carefully.

Alternative Formatting Methods

The following code fragments are equivalent means of specifying the same output format. They show how I, F and E *edit specifiers* display the number 55 in integer, fixed-point and floating-point formats.

(i) Using a FORMAT statement with a label (here 150):

```
WRITE( *, 150 ) 55, 55.0, 55.0
...
150 FORMAT( 1X, I3, 1X, F5.2, 1X, E8.2 )
```

(ii) Putting the format directly into the WRITE statement:

```
WRITE( *, '( 1X, I3, 1X, F5.2, 1X, E8.2 )' ) 55, 55.0, 55.0
```

(iii) Putting the format in a character variable C (either in its declaration as here, or a subsequent assignment):

```
CHARACTER (LEN=*), PARAMETER :: C = '( 1X, I3, 1X, F5.2, 1X, E8.2 )'
...
WRITE( *, C ) 55, 55.0, 55.0
```

Any of these will output (to the screen):

```
55 55.00 0.55E+02
```

Terminology

A *record* is an individual line of input/output.

A *format specification* describes how data is laid out in (one or more) records.

A *label* is a number in the range 1-99999 preceding a statement on the same line. The commonest uses are in conjunction with FORMAT statements and to indicate where control should pass following an i/o error.

Edit Descriptors

A *format specification* consists of a series of *edit descriptors* (e.g. I4, F7.3) separated by commas and enclosed by brackets. The commonest edit descriptors are:

Iw	integer in a field of width <i>w</i> ;
Fw.d	real, fixed-point format, in a field of width <i>w</i> with <i>d</i> decimal places;
Ew.d	real, floating-point (<i>exponential</i>) format in a field of width <i>w</i> with <i>d</i> decimal places;
nPEw.d	floating point format as above with <i>n</i> significant figures in front of the decimal point;
Lw	logical value (T or F) in a field of width <i>w</i> ;
Aw	character string in a field of width <i>w</i> ;
A	character string of length determined by the output list;
'text'	a character string actually placed in the format specification;
nX	<i>n</i> spaces
Tn	move to position <i>n</i> of the current record;
/	start a new record.

This is only a fraction of the available edit descriptors – see the recommended textbooks.

Notes:

- (1) If the required number of characters is less than the specified width then the output will be right-justified in its field.
- (2) (For numerical output) if the required number of characters exceeds the specified width then the field will be filled with asterisks. E.g, attempting to write 999 with edit descriptor I2 will result in **.
- (3) Attempting to write output using an edit specifier of the wrong type (e.g. 3.14 in integer specifier I4) will result in a run-time error – try it so that you recognise the error message.
- (4) The format specifier will be used repeatedly until the output list is exhausted. Each use will start a new record. For example,

```
WRITE( *, '( 1X, I2, 1X, I2, 1X, I2 )' ) ( I, I = 1, 5 )
```

will produce the following lines of output:

```
1 2 3  
4 5
```
- (5) If the whole format specifier isn't required (as in the last line above) the rest is simply ignored.

Repeat Counts

Format specifications can be simplified by collecting repeated sequences together in brackets with a repeat factor. For example, the code example above could also be written

```
WRITE( *, '( 3( 1X, I2 ) )' ) ( I, I = 1, 5 )
```

Historical Baggage: Carriage Control

It is recommended that the first character of an output record be a blank. This is best achieved by making the first edit specifier either 1X (one blank space) or T2 (start at the second character of the record). In the earliest versions of Fortran the first character effected line control on a line printer. A blank meant 'start a new record'. Although such carriage control is long gone, some i/o devices may still ignore the first character of a record.

10.4 The READ Statement

In the input statement

```
READ( unit, format ) list
```

list is a set of variables to receive the data. *unit* and *format* are as for the corresponding WRITE statement.

Although formatted reads are possible (an example is given in the example programs for Week 4), it is uncommon for *format* to be anything other than * (i.e. list-directed input). If there is more than one variable in *list* then the input items can be separated by blank spaces, commas or simply new lines.

Notes.

- (1) Each READ statement will keep reading values from the input until the variables in *list* are assigned to, even if this means going on to the next record.
- (2) Each READ statement will, by default, start reading from a new line, even if there is input data unread on the previous line. In particular, the statement

```
READ( *, * )
```

(with no *list*) will simply skip a line of unwanted data.
- (3) The variables in *list* must correspond in type to the input data – there will be a run-time error if you try to read a number with a decimal point into an integer variable, or some text into a real variable, for example.

Example. The following program reads the first two items from each line of an input file `input.dat` and writes their sum to a file `output.dat`.

```
PROGRAM IO
  IMPLICIT NONE
  INTEGER I
  INTEGER A, B

  OPEN( 10, FILE='input.dat' )
  OPEN( 20, FILE='output.dat' )

  DO I = 1, 4
    READ( 10, * ) A, B
    WRITE( 20, * ) A + B
  END DO

  CLOSE( 10 )
  CLOSE( 20 )
END PROGRAM IO
```

A sample input file (`input.dat`) is:

```
10 3
-2 33
3 -6
40 15
```

Exercise:

- (1) Type the source code into file `io.f95` (say) and the input data into `input.dat` (saving it in the same folder). Compile and run the program and check the output file.
- (2) Modify the program to write the output to screen instead of to file.
- (3) Modify the program to format the output in a tidy column.
- (4) Try changing the input file and predicting/observing what happens. Note any run-time error messages.
 - (i) Change the first data item from 10 to 10.0 – why does this fail at run-time?
 - (ii) Add an extra number to the end of the first line – does this make any difference to output?
 - (iii) Split the last line with a line break – does this make any difference to output?
 - (iv) Change the loop to run 3 times (without changing the input data).
 - (v) Change the loop to run 5 times (without changing the input data).

10.5 Repositioning Input Files

`REWIND unit` repositions the file attached to *unit* at the first record.
`BACKSPACE unit` repositions the file attached to *unit* at the start of the previous record.

Obviously, neither will work if *unit* is attached to the keyboard!

10.6 Additional Specifiers

The general form of the `READ` statement is

```
READ( unit, format[, specifiers] )
```

Some useful specifiers are:

`IOSTAT = integer-variable` assigns *integer-variable* with a number indicating status
`ERR = label` jump to *label* on an error (e.g. missing data or data of the wrong type);
`END = label` jump to *label* when the end-of-file marker is reached.

`IOSTAT` returns zero if the read is successful, different negative integers for end-of-file (EOF) or end-of-record (EOR), and positive integers for other errors. (For FTN95, -1 means EOF and -2 means EOR: this isn't guaranteed by the Fortran standard, but it seems to work for all the compilers available at the University)

Non-Advancing Input/Output

By default, each READ or WRITE statement will automatically conclude with a carriage return/line feed. This can be prevented with an ADVANCE = 'NO' specifier; e.g.

```
WRITE( *, '(A)', ADVANCE = 'NO' ) 'Enter a number: '  
READ *, I
```

Note that a format specifier (here, '(A)') must be used for non-advancing i/o, even for a simple output string. The following statement won't work:

```
WRITE( *, *, ADVANCE = 'NO' ) 'Enter a number: '
```

Assuming CH has been declared as a character variable (of length 1) then a single character at a time can be read from a text file attached to unit 10 by:

```
READ( 10, '( A1 )', IOSTAT = IO, ADVANCE = 'NO' ) CH
```

To see programs using additional specifiers see the example programs for week 4.

11. MODULES

See Sample Programs – Week 4

Modules are the fourth type of program unit (after *main programs*, *subroutines* and *functions*). They were new in Fortran 90 and typify modern programming practice.

A module has the form:

```
MODULE module-name
  type declarations
  [ CONTAINS
    internal subprograms ]
END [MODULE [ module-name ] ]
```

The main uses of a module are:

- to allow sharing of variables between multiple program units;
- to collect together related internal subprograms (functions or subroutines).

Other program units have access to these module variables and internal subprograms via the statement

```
USE modulename
```

(which should be placed at the start of the program unit, before any `IMPLICIT NONE` or type statements).

Modules make redundant older elements of Fortran such as `COMMON` blocks (used to share variables), statement functions (one-line internal functions) and the (never-actually-standarised) `INCLUDE` statement.

11.1 Sharing Variables

Variables are passed between one program unit and another via argument lists. For example a program may call a subroutine `POLARS` by

```
CALL SUBROUTINE POLARS( X, Y, R, THETA )
```

The program passes `X` and `Y` to the subroutine and receives `R` and `THETA` in return. Any other variables declared in one program unit are completely unknown to the other, even if they have the same name. Other routines may also call `POLARS` in the same way.

Communication by argument list alone is OK provided the argument list is (a) short; and (b) unlikely to change with code development. Problems arise if:

- a large number of variables need to be shared between program units; (the argument list becomes long and unwieldy);
- code is under active development; (the variables being shared may need to be changed, and will have to be changed consistently in every program unit making that subroutine call).

Modules solve these problems by maintaining a central collection of variables which can be modified when required. Any changes only need to be made in the module. This is analogous to a web-based database – changes are made in one place (the server), but can be made from anywhere linked to it.

11.2 Internal Functions

Subroutines and functions can be *external* or can be *internal* to bigger program units. However, internal subprograms are accessible only to the program unit in which they are defined (which is a bit selfish!). Thus, they are only of use for short, simple functions specific to that program unit. A better vehicle for an internal function is a module, because if that contains internal functions or subroutines then they are accessible to all the subprograms that `USE` that module.

11.3 Example

The following, somewhat trite, example illustrates how a program (CONE) uses a routine from a module (GEOMETRY) to find the volume of a cone. Note the USE statement in the main program and the structure of the module. The arrangement is convenient because if we later decide to add a new global variable or geometry-related function (say, a function VOLUME_CYLINDER) it is easy to do so within the module.

```
MODULE GEOMETRY
! Functions to compute areas and volumes
  IMPLICIT NONE

  ! Shared variables
  REAL, PARAMETER :: PI = 3.14159

CONTAINS
  ! Internal subprograms

  REAL FUNCTION AREA_CIRCLE( R )
    REAL R
    AREA_CIRCLE = PI * R ** 2
  END FUNCTION AREA_CIRCLE

  REAL FUNCTION AREA_TRIANGLE( B, H )
    REAL B, H
    AREA_TRIANGLE = 0.5 * B * H
  END FUNCTION AREA_TRIANGLE

  REAL FUNCTION AREA_RECTANGLE( W, L )
    REAL W, L
    AREA_RECTANGLE = W * L
  END FUNCTION AREA_RECTANGLE

  REAL FUNCTION VOLUME_SPHERE( R )
    REAL R
    VOLUME_SPHERE = (4.0 / 3.0) * PI * R ** 3
  END FUNCTION VOLUME_SPHERE

  REAL FUNCTION VOLUME_CUBOID( W, L, H )
    REAL W, L, H
    VOLUME_CUBOID = W * L * H
  END FUNCTION VOLUME_CUBOID

  REAL FUNCTION VOLUME_CONE( R, H )
    REAL R, H
    VOLUME_CONE = PI * R ** 2 * H / 3.0
  END FUNCTION VOLUME_CONE

END MODULE GEOMETRY
```

```
PROGRAM CONE
  USE GEOMETRY           ! declare use of module
  IMPLICIT NONE
  REAL RADIUS, HEIGHT

  PRINT *, 'Input radius and height'
  READ *, RADIUS, HEIGHT

  PRINT *, 'Volume: ', VOLUME_CONE( RADIUS, HEIGHT )
END PROGRAM CONE
```

Note that the internal functions of the module (as well as any program units using the module) automatically have access to any module variables above the CONTAINS statement (here, just `PI`).

11.4 Compiling Programs With Modules

The module may be in the same source file as other program units or it may be in a different file. To enable the compiler to operate correctly, however, the module must be compiled before any program units that USE it. Hence,

- if it is in a different file the module must be compiled first;
- if it is in the same file the module must come before any program units that USE it.

Compilation results in a special file with the same root name and filename extension `.mod` and, if there are internal subprograms, an accompanying `.obj` file.

Assuming that the program is in file `cone.f95` and the module in file `geometry.f95` the compilation and linking commands for the above example (with the FTN95 compiler) are

```
ftn95 geometry.f95
ftn95 cone.f95
slink cone.obj geometry.obj
```

This will create an executable file `cone.exe` (based on the first name supplied to the linker).

If running from the command window then the sequence of compiling and linking commands can conveniently be put in a batch file.

12. OTHER FORTRAN FEATURES

This course is too short to cover more advanced features of Fortran. However, you may like to investigate for yourself some of the following:

- *derived data types*
- *pointers*
- *object-oriented programming* (Fortran 2003 only)

12.1 Derived Data Types

The intrinsic data types are *integer*, *real*, *complex*, *character*, *logical*. However, you are at liberty to devise your own *composite* or *derived* data types. These are like database records.

For example you might define a type STUDENT as:

```
TYPE STUDENT
  CHARACTER (LEN=20) SURNAME
  CHARACTER (LEN=20) FIRSTNAME
  INTEGER YEAR
  REAL MARK
END TYPE
```

Variables A, B, C could then be declared of this type by

```
TYPE (STUDENT) A, B, C
```

Such variables are also called *structures*. Note that each one has multiple components, which are obtained by use of the *% component selector* (see below).

Variables of derived type can be assigned either by specifying individual components; e.g.

```
A%SURNAME = 'Apsley'
A%FIRSTNAME = 'David'
A%YEAR = 2
A%MARK = 75.0
```

or, more efficiently, as a single statement:

```
A = STUDENT( 'Apsley', 'David', 2, 75.0 )
```

(The second method can also be included in the type declaration).

They can then be used as single entities, or via their individual components; e.g.

```
IF ( A%MARK > 69.5 ) PRINT *, 'First class'
```

You can have arrays of structures, just like arrays of any other type; e.g.

```
TYPE (STUDENT), DIMENSION(300) :: CIVIL_ENGINEERING
```

12.2 Pointers

Pointer variables (with type declaration including the *POINTER* attribute) can be used to “point to” other variables (declared with the *TARGET* attribute). They thus form an alias for that target variable; the target can change during program operation. Pointers are also useful in linked lists – you don’t need to continually re-sort arrays, just have pointers to the next element.

12.3 Object-Oriented Programming (Fortran 2003 only)

Object-oriented programming is a modern paradigm or style of programming that includes the concepts of:

- *encapsulation* (“self-contained; private”) – individual *objects* of a given *class* contain variables (“*properties*”) and code that are hidden to the outside world and can only be accessed indirectly by subprograms (“*methods*”) of that object. The best analogy I can think of is the Faculty’s IT support object where the individual computer officers constitute the properties, are hidden from view in a keypad-accessible office off the Pariser computer cluster, and can only be accessed by the “Hotline”

method!

- *modularity* (“small, specialised parts”) – breaking down a complex task into many small, specific and reusable individual tasks, which are only incorporated when needed. This is the purpose of subroutines and functions.
- *polymorphism* (“same name, but operation depends on context”) – allows calling a routine of the same name with arguments of different type; e.g. the *SQRT* function with arguments which may be real or complex, scalar or array. In Fortran this also includes *operator overloading* so that you could (if you really wanted to) redefine the * operator to mean “subtraction” when applied to certain types of variables. Hmm.
- *inheritance* – a subclass automatically receives the properties and methods of its parent. As a real-world analogy a class “Road” may have subclasses “Motorway” and “Street”.

As you might infer, I am not a great fan of object-oriented programming, which is the *raison d’être* of languages like C++, but seems to lead to unjustified complexity for most engineering tasks. Chapman’s textbook explains it well in the Fortran context. Note that there are precious few (if any) compilers that fully implement the Fortran 2003 standard.

APPENDICES

A1. Order of Statements in a Program Unit

If a program unit contains no internal subprograms then the structure of a program unit is as follows.

PROGRAM, FUNCTION, SUBROUTINE or MODULE statement		
USE statements		
FORMAT statements	IMPLICIT NONE statement	
	PARAMETER and DATA statements	type declarations
	executable statements	
END statement		

Where internal subprograms are to be used, a more general form would look like:

PROGRAM, FUNCTION, SUBROUTINE or MODULE statement		
USE statements		
FORMAT statements	IMPLICIT NONE statement	
	PARAMETER and DATA statements	type declarations
	executable statements	
CONTAINS		
internal subprograms		
END statement		

A2. Fortran Statements

The following list is of the more common statements and is not exhaustive. A more complete list may be found in the recommended textbooks. To dissuade you from using them, the table does not include elements of earlier versions of Fortran – e.g. COMMON blocks, DOUBLE PRECISION real type, INCLUDE statements, CONTINUE and (the truly awful!) GOTO – whose functionality has been replaced by better elements of Fortran 90/95.

ALLOCATE	Allocates dynamic storage.
BACKSPACE	Positions a file before the preceding record.
CALL	Invokes a subroutine.
CASE	Allows a selection of options.
CHARACTER	Declares character data type.
CLOSE	Disconnects a file from a unit.
COMPLEX	Declares complex data type.
CONTAINS	Indicates presence of internal subprograms.
DATA	Used to initialise variables at compile time.
DEALLOCATE	Releases dynamic storage.
DIMENSION	Specifies the size of an array.
DO	Start of a repeat block.
DO WHILE	Start of a block to be repeated while some condition is true.
ELSE, ELSE IF, ELSE WHERE	Conditional transfer of control.
END	Final statement in a program unit or subprogram.
END DO, END IF, END SELECT	End of relevant construct.
EQUIVALENCE	Allows two variables to share the same storage.
EXIT	Allows exit from within a DO construct.
EXTERNAL	Specifies that a name is that of an external procedure.
FORMAT	Specifies format for input or output.
FUNCTION	Names a function subprogram.
IF	Conditional transfer of control.
IMPLICIT NONE	Suspends implicit typing (by first letter).
INTEGER	Declares integer type.
LOGICAL	Declares logical type.
MODULE	Names a module.
OPEN	Connects a file to an input/output unit.
PRINT	Send output to the standard output device.
PROGRAM	Names a program.
READ	Transfer data from input device.
REAL	Declares real type.
RETURN	Returns control from a subprogram before hitting the END statement.
REWIND	Repositions a sequential input file at its first record.
SELECT CASE	Transfer of control depending on the value of some expression.
STOP	Stops a program before reaching the END statement.
SUBROUTINE	Names a subroutine.
TYPE	Defines a derived type.
USE	Enables access to entities in a module.
WHERE	IF-like construct for array elements.
WRITE	Sends output to a specified unit.

A3. Type Declarations

Type statements:

INTEGER
REAL
COMPLEX
LOGICAL
CHARACTER
TYPE (user-defined, derived types)

The following attributes may be specified.

ALLOCATABLE
DIMENSION
EXTERNAL
INTENT
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
TARGET

Variables may also have a KIND, which will affect the numerical precision with which they are stored.

A4. Intrinsic Routines

A comprehensive list can be found in the recommended textbooks or in the compiler's help files.

Mathematical Functions

(Arguments X, Y etc. can be real or complex, scalar or array unless specified otherwise)

COS(X), SIN(X), TAN(X) – trigonometric functions (arguments are in *radians*)
ACOS(X), ASIN(X), ATAN(X) – inverse trigonometric functions
ATAN2(Y, X) – inverse tangent of Y/X in the range $-\pi$ to π (real arguments)
COSH(X), SINH(X), TANH(X) – hyperbolic functions
EXP(X), LOG(X), LOG10(X) – exponential, natural log, base-10 log functions
SQRT(X) – square root
ABS(X) – absolute value (integer, real or complex)
MAX(X1, X2, ...), MIN(X1, X2, ...) – maximum and minimum (integer or real)
MODULO(X, Y) – X modulo Y (integer or real)
MOD(X, Y) – remainder when X is divided by Y

Type Conversions

INT(X) – converts real to integer type, truncating towards zero
NINT(X) – nearest integer
CEILING(X), FLOOR(X) – nearest integer greater than or equal, less than or equal
REAL(X) – convert to real
CMPLX(X) or CMPLX(X, Y) – real to complex
CONJG(Z) – complex conjugate (complex Z)
AIMAG(Z) – imaginary part (complex Z)
SIGN(X, Y) – absolute value of X times sign of Y

Character-Handling Routines

CHAR(I) – character in position I of the system collating sequence;
ICHAR(C) – position of character C in the system collating sequence.
ACHAR(I) – character in position I of the ASCII collating sequence;
IACHAR(C) – position of character C in the ASCII collating sequence.

LEN(STRING) – declared length of STRING, even if it contains trailing blanks;
 TRIM(STRING) – same as STRING but without any trailing blanks;
 LEN_TRIM(STRING) – length of STRING with any trailing blanks removed.

ADJUSTL(STRING) – left-justified STRING
 ADJUSTR(STRING) – right-justified STRING

INDEX(STRING, SUBSTRING) – position of first occurrence of SUBSTRING in STRING
 SCAN(STRING, SET) – position of first occurrence of *any* character from SET in STRING
 VERIFY(STRING, SET) – position of first character in STRING that is *not* in SET

Array Functions

DOT_PRODUCT(vector_A, vector_B) – scalar product (integer or real)
 MATMUL(matrix_A, matrix_B) – matrix multiplication (integer or real)
 TRANSPOSE(matrix) – transpose of a matrix
 MAXVAL(array), MINVAL(array) – maximum and minimum values (integer or real)
 PRODUCT(array) – product of values (integer, real or complex)
 SUM(array) – sum of values (integer, real or complex)

A5. Operators

Numeric Intrinsic Operators

<i>Operator</i>	<i>Action</i>	<i>Precedence (1 is highest)</i>
**	Exponentiation	1
*	Multiplication	2
/	Division	2
+	Addition or unary plus	3
-	Subtraction or unary minus	3

Relational Operators

<i>Operator</i>	<i>Operation</i>
< or .LT.	less than
<= or .LE.	less than or equal
== or .EQ.	equal
/= or .NE.	not equal
> or .GT.	greater than
>= or .GE.	greater than or equal

Logical Operators

<i>Operator</i>	<i>Action</i>	<i>Precedence (1 is highest)</i>
.NOT.	logical negation	1
.AND.	logical intersection	2
.OR.	logical union	3
.EQV.	logical equivalence	4
.NEQV.	logical non-equivalence	4

Character Operators

// concatenation