FORTRAN AUTUMN 2005

Recommended Books

Hahn, B.D., 1994, *Fortran 90 For Scientists and Engineers*, Arnold
Smith, I.M., 1995, *Programming in Fortran 90. A First Course For Engineers and Scientists*, Wiley

1. BACKGROUND TO FORTRAN

1.1 Terminology

A *computer* is a machine capable of storing and executing sets of instructions, called *programs*, in order to solve specific problems.

A *platform* refers to the combination of computer + operating system.

A *programming language* is a particular set of rules (with its own grammar or *syntax*) for coding the instructions to a computer.

Source code (human-readable) is converted to *binary code* (computer-readable) in the process of *compilation*. This is achieved by running a special program called a *compiler*.

A *high-level* programming language. (e.g. Fortran, C, Pascal, Java,…) is human-comprehensible and capable of running on any platform with a suitable compiler. A *low-level* language (like assembler) is machine-dependent and makes direct instructions to the processor.

1.2 Fortran History

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

Fortran is highly-*standardised*, making it extremely *portable* (able to run under 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
Fortran 90 – ISO/TEC 1539:1991
Fortran 95 – ISO/IEC 1539-1: 1997 – a (very) minor revision of Fortran 90

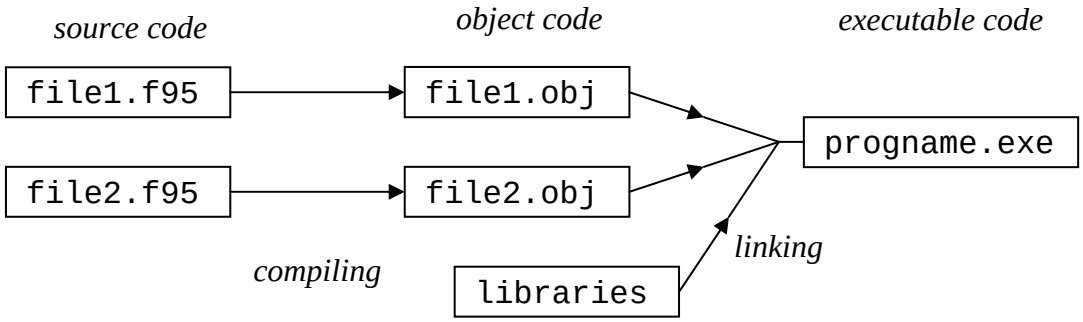
The compiler which we shall use is provided by Salford Software and happens to be a Fortran 95 compiler. However, everything we do will actually conform to the Fortran 90 standard.

Salford Fortran – like many Fortran implementations – comes with additional *library* routines for producing, for example, graphical output or Windows applications.

1.3 Creating Executable Code – General Procedure

For all high-level languages (Fortran, C, Pascal, …) producing executable code is a two-stage process:

(i) *Compiling* converts source code to binary *object* code.
(ii) *Linking* combines one or more files of compiled code with additional library routines to create an *executable* program.



Source code is a human-readable set of instructions that can be created and modified on any computer with any text editor. It consists of one or more files.

Fortran files typically have filetype `.f90` or `.f95`.

C++ files typically have filetype `.CPP`

Each source file is compiled (by a special program called a *compiler*) to create a corresponding *object code* file. These are computer-readable and platform-dependent.

On a PC object code usually has filetype `.obj`

One or more object code files are linked (by a special program called a *linker*) with any required library routines to create a single *executable* program. On a PC this would have filetype `.exe`

Most Fortran codes consist of multiple *subprograms*, all performing specific, independent functions. Different sets of subprograms may be contained in different source files which must be compiled separately and then linked. The advantages of having collections of routines in different files is that it is easy to re-use subprograms in different applications. Many important subprograms are kept together as pre-compiled *libraries*. Examples in engineering are the NAG (National Algorithms Group) libraries for mathematical programming or Salford's ClearWin libraries for creating Windows applications.

1.4 Creating Executable Code – Salford Fortran

Source code can be created with any text editor. In the University clusters, suitable editors are:

`notepad` – supplied with the Windows operating system

`plato` – supplied with the Salford software

but many other editors are available.

The precise commands used to compile and link will depend on the particular platform and compiler. For the Salford Fortran 95 compiler on a PC the relevant programs are:

compiler: `ftn95`

linker: `slink`

In the Salford implementation, files containing Fortran source code should have filetype `.f95` or `.f90`.

Salford's Fortran implementation (like many others) includes additional applications to facilitate program development:

- an *integrated development environment* or graphical interface (`plato`);
- additional *library routines* (`ClearWin+` for writing Windows interfaces);
- a *debugging* facility (`sdbg`).
- a *make* facility for better control of compiling and linking.

Only the first of these will be covered in this course, but the rest are available and worth investigating if you intend to pursue Fortran programming further.

2. RUNNING A FORTRAN PROGRAM

You have TWO options:
(i) Use the Command Window
(ii) Use an "integrated development environment" (`plato2`)

Those who have grown up with Windows will probably find the latter more friendly, but it tends to obscure the basic processes going on and is Salford-Fortran-specific, so we will examine both options.

2.1 Salford Fortran in the University Clusters

The Salford Software programs group can be accessed from the usual `Start` menu:

```
Start > All Programs
      > Programs - Core
      > Compilers
      > Salford Software
      > Salford FTN95 Command-line Environment     or     Plato2 IDE
```

(Although there are plenty of other ways of starting a Command Window, starting from the Salford Software link should ensure that the PCs in the cluster are able to find all the necessary compilers and run-time libraries.)

2.2 Using the Command Window

(A brief summary of the more common commands in the Command Window can be found in the Internet resources for this course.)

Open a command window as in Section 2.1.

Navigate to, e.g., your p: drive (if necessary):

```
p:
```

Create a directory (aka "folder") to put your work in:

```
md myfortran
```

Then change to that directory:

```
cd myfortran
```

Create the following simple source file with any editor, e.g., `notepad`:

```
notepad prog1.f95
```

(Notepad will tell you if the file doesn't already exist and ask you to confirm creation).

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

Make sure that you save the file.

Compile the code by entering the command

```
ftn95 prog1.f95
```

This will (by default) create the binary object file `prog1.obj`.

Link the code by entering the command

```
slink prog1.obj
```

This will (again by default) create an executable file `prog1.exe`

Run the program by entering the command

```
prog1
```

Various options can be passed to the compiler. These vary considerably between Fortran compilers. Typical examples for Salford Fortran are given below.

`ftn95 prog1.f95 /link`
– invokes the linker immediately after compiling.

`ftn95 prog1.f95 /full_debug /undef`
– debugging options; useful during development but will slow down final code.

`ftn95 /help`
– brings up a (moderately useful) help system, including the complete set of compile options.

2.3 Using Salford's Integrated Development Environment (`Plato`)

Open the `Plato` integrated development environment as in Section 2.1.

An integrated development environment (IDE) is basically there to assist in program development. It consists of an advanced text editor with all the buttons you need to carry out compiling/linking/executing (plus a lot of other things) with the click of a mouse. It can optionally provide "syntax highlighting", i.e. colouring sections of code according to their function: keyword, variable, comment etc. This occasionally helps.

(Note, however, that `plato` is specific to Salford software and if the University ever goes over to another compiler then you are stuffed! Hence we teach the Command Window version as well.)

Type in the same source file.

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

Save the source code (in any folder of your choice) as `prog2.f95` (The `.f95` extension is vital!)

Compile the code by using the pull-down menu
`Project > Compile file`
(You will find a handy little button on the toolbar that does exactly the same thing.)
This will create an object file `prog2.obj`.

Compile and link ("build") the code by using the pull-down menu
`Project > Build file`
(Again, you will find a handy little button on the toolbar that does exactly the same thing.)
This will create an executable file `prog2.exe`

Run the program by either:
Hitting the `RUN` button after you have complied and linked the code
or
Using the pull-down menu: `Project > Run`
or
Clicking the appropriate button on the toolbar

Actually, using `Project > Run` will automatically
save,
compile,
link,
run
the code. However, it is usually better at first to do these in separate steps, so that you can debug your program.

If the compiler encounters any mistakes then it will list these in an errors window and you can go direct to the appropriate line of code by clicking on the particular error.

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'.

(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 `*` indicates 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). Later we shall see 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.

(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 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 not true in some programming languages, notably C and C++, so it is probably best not to get in the habit of doing it.

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 widely used to clarify program structure. Typical use is 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 (see later) by a similar amount.

(11) Running the Program.

Follow the instructions in the first section to compile and link the program. Run it by entering its name at the command prompt or from within `Plato`. 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 Building Blocks of the Language

The Fortran *character set* consists of:

the *alphanumeric* characters: A, …, Z, a, …, z, 0, …, 9 and _(*underscore*)
the special symbols: (*blank*) = + - * / ( ) , . ' $: " % & ; < > ?

From the character set we can build *tokens* which are one of six types:

labels	e.g.	`100  1234  9999`
constants	e.g.	`15  30.5  'This is a string'  .TRUE.`
keywords	e.g.	`PROGRAM  END  IF  DO`
names	e.g.	`MYNAME  Manchester_United  Chelsea123`
operators	e.g.	`+  -  *  /  **  >`
separators	e.g.	`(  )  :  ;`

From tokens we can build *statements*. e.g.

```
X = ( -B + SQRT( B ** 2 – 4.0 * A * C ) ) / ( 2.0 * A )
```

From statements we can build *program units*.

4.2 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 an 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!`	(`!` is not allowed)

4.3 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.

In advanced applications 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 (signed or unsigned) 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). 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.`

4.4 Declaration of Variables

Type Declarations

Variables should be *declared* (that is, have their 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 NUMBER_OF_PEOPLE
REAL RESULT
COMPLEX Z
LOGICAL ANSWER
```

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

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

Initialisation

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

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

Variables can also be initialised at compile time with a `DATA` statement; e.g.

```
DATA NUMBER_OF_PEOPLE, RESULT, 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
```

The double colon (`::`) must be used when attributes are specified.

Kind (*Optional*)

The *kind* concept will not be mentioned much in this course, but it is valuable in ensuring true portability across platforms and one should be aware of its existence. Basically, the default memory size and format of storage for the various data types is not set by the standard and varies between Fortran implementations – for example 2 or 4 bytes for an integer, 4 or 8 bytes for a real. This affects both the largest integer that can be represented and the accuracy with which real numbers can be stored. If you wish true portability then you may wish to declare the *kind type parameter* 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 declare all integers and reals with the required range and precision.

To print out the default kind types for the Salford Fortran 95 compiler, try

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

where the intrinsic function `KIND` returns the kind type of its argument: in this case integer and real values.

The `kind` parameter will not be used in this introductory course, but is described in the recommended books.

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. 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. To admit older standards as a subset, Fortran has to go on doing this. However, it is appalling programming practice 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 (the compiler will then flag any variable that you have forgotten to declare).

4.5 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.

operator	*meaning*	*precedence* (1 = highest)
`**`	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	$1 + (2 \times 3)$ or 7
`10.0 / 2.0 * 5.0`	evaluates as	$(10.0 / 2.0) \times 5.0$ or 25.0
`5.0 * 2.0 ** 3`	evaluates as	$5.0 \times (2.0^3)$ or 40.0

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

`A ** B ** C` evaluates as A^{B^C}

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` → `3.0 /10.0` → `0.3`

The biggest 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. This time, however, the conversion is to the type of the variable being assigned. Suppose `I` is an integer. Then 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`.

4.6 Character Operators

There is only one character operator, *concatenation*, //:

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

4.7 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.` → `.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.

4.8 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.

- 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.

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

  is the same as the single-line statement

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

There may be up to 132 characters per line. However, editor defaults (and historical limits in previous versions of Fortran) mean that most programmers do not use lines longer than 72 characters.

5. REPETITION: `DO` AND `DO WHILE`

See Sample Programs – Week 2

One advantage of computers is that they never get bored by repeating the same action many times. For example, consider the following program.

```
PROGRAM LINES
! Illustration of DO-loops
   IMPLICIT NONE

   INTEGER L                                  ! a counter

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

END PROGRAM LINES
```

This illustrates how a `DO` loop may be used to carry out the same statement or set of statements many times. The main forms of loop structure are:

(i) Deterministic `DO` loop – the maximum number of loops is specified:

```
DO variable = expression1, expression2 [, expression3]
      repeated section
END DO
```

(ii) Non-deterministic `DO` loop: `EXIT` the loop when some criterion is met.

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

(iii) Alternative form of non-deterministic loop.

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

In the first of these, *variable* is an integer variable to be used as a loop counter and *expression1*, *expression2*, *expression3* are integers or, more generally, integer expressions. *expression1* and *expression2* are the limits of the count and *expression3* is the increment (which may be positive or negative). If *expression3* is not specified, it is assumed to be 1. If *expression3* is positive then the loop will stop executing once the integer variable exceeds *expression2*.

In the last two examples, looping stops when some logical criterion is met.

DO loops can be *nested* (i.e. one inside another). Indentation is definitely recommended here. For example:

```
PROGRAM NESTED
! Illustration of nested DO-loops
   IMPLICIT NONE

   INTEGER I, J                                  ! loop counters

   DO I = 1, 6                                   ! start of outer loop
      PRINT *, 'Outer loop with I = ', I
      DO J = 1, 3                                ! start of inner loop
         PRINT *, ' I, J = ', I, J
      END DO
      PRINT *                                    ! a blank line
   END DO                                        ! end of repeated section

END PROGRAM NESTED
```

The DO loop counter should be an integer. To increment in a non-integer sequence, e.g. 0.5, 0.8, 1.1, ... , define separate loop counters (e.g. I), increment (e.g. DX), initial value (e.g. X0) and for each pass of the loop work out the value to be output, as in the example below:

```
PROGRAM XLOOP
! Illustration of non-integer values
   IMPLICIT NONE

   INTEGER I                                     ! loop counter
   REAL DX                                       ! increment
   REAL X0                                       ! non-integral initial value
   REAL X                                        ! value to be output

   X0 = 0.5                                      ! set initial value
   DX = 0.3                                      ! set increment

   DO I = 1, 10                                  ! start of repeated section
      X = X0 + (I - 1) * DX                      ! actual 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
```

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.

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 often 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. CASE is often more efficient than an IF ... ELSE IF ... ELSE construct because only one expression need be evaluated.

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}, ..., 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), … , (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

N	
x_1	y_1
x_2	y_2
...	
x_N	y_N

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

```
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 Memory Allocation

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 things – all of them garbage and 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
```

Most departmental secretaries will not be impressed with this error message.

A far better solution is to use *dynamic memory allocation*; that is, the array size is determined (and memory space 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 size of the arrays is not specified, but is replaced by a single colon (`:`).

(ii) When the arrays are needed, 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 or commas) which it encounters. Since all the points are read in one go, they no longer need to be on separate lines of the input file.

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.

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 Σx_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

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

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 civil 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) = A(1,1) + A(1,3)
   etc.
```

Terminology

dimension – a particular subscript
rank – the number of subscripts (1 for a vector, 2 for a matrix etc.)
extent – the number of elements in a particular dimension
shape – the set of extents

For example, the declaration

```
REAL X(0:100,3,3)
```

declares:

- an array of real type
- named X
- of rank 3
- of extent 101 along the first, 3 along the second and 3 along the third dimension
- of shape 101×3×3

A typical element is, e.g., `X(50,2,2)`.

Matrix Multiplication

Matrix multiplication can be accomplished by nested `DO` loops (see below). However, Fortran provides an intrinsic function `MATMUL` to do the same in a single statement.

Consider the matrix multiplication `C=AB`, where `A`, `B` and `C` are 3×3 matrices declared by

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

A nested `DO` loop construct can be used to evaluate the product; for example,

```
DO I = 1, 3
   DO J = 1, 3
      C(I,J) = A(I,1) * B(1,J) + A(I,2) * B(2,J) + A(I,3) * B(3,J)
   END DO
END DO
```

However, the multiplication can also be accomplished by the single statement

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

Reasonable?

Note that, for matrix multiplication to be legitimate, the number of columns in `A` must equal the number of rows in `B`; i.e. the matrices are *conformable*.

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 so that, 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 and Y are declared with 10 elements:

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

Assignment

```
X = 10.0
```

sets every element of X to the value 10.0.

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.

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 engineering calculations, not word-processing. However, it now has 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, as it is the clearest to read).

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 and EBCDIC. 'Less than' (<) and 'greater than' (>) refer to the position of the characters in this collating sequence.

The Fortran standard requires that upper-case letters `A-Z` and lower-case letters `a-z` are separately in alphabetical order, and 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:

```
'Dr' > 'Apsley'
'1st year' < '2nd year'
```

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.

```
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 *any* character from `SET` in `STRING`
`VERIFY( STRING, SET )`	position of first character in `STRING` that is *not* 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 certain parameters. 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*

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 entered in 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, Salford Fortran 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 common 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

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

```
[PROGRAM [name]]
    USE statements
    [IMPLICIT NONE]
    type declarations
    executable statements
END [PROGRAM [name]]
```

Subroutines

```
SUBROUTINE name (argument-list)
    USE statements
    [IMPLICIT NONE]
    type declarations
    executable statements
END [SUBROUTINE [name]]
```

Functions

```
[type] FUNCTION name (argument-list)
    USE statements
    [IMPLICIT NONE]
    type declarations
    executable statements
END [FUNCTION [name]]
```

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)

```
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
```

(Using a subroutine)

```
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
```

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, 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 unit. 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*.

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 :: VALUE
```

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 Hahn, 1994). Some of the more useful ones are given in Appendix A4.

9.7 Modules

See Sample Programs – Week 4

Modules are used to:

- make a large number of variables accessible to several program units without the need for a large and unwieldy argument list;
- collect together related internal subprograms.

A module has the form:

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

Each module's source code should be placed in its own `.f95` file and compiled before any program which `USE`s it. Compilation results in a special file with the same root name and filename extension `.mod` . It is then made accessible to any main program or subprogram by the statement

`USE` *module-name*

All variables, executable code and internal subprograms in the module are then made available to any program unit which `USE`s this module.

A particular advantage is that changes during program development to the set of global variables used need only be made in one source file rather than in numerous program units and argument lists. Modules, which were introduced with Fortran 90, make the `INCLUDE` statements and `COMMON`-block features of earlier versions of Fortran redundant.

10. ADVANCED INPUT/OUTPUT

See Sample Programs – Week 4

Hitherto we have used *list-directed* input/output (i/o) with the 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.

10.1 READ and WRITE

General 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 1-99 which has been *attached* (see below) to a particular i/o device;
- 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.

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 the `OPEN` statement. e.g.

```
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, Hahn (1994). The general form of the `OPEN` statement for reading or writing a non-temporary file is

```
OPEN ([UNIT = ]unit, FILE = file[, specifiers] )
```

There may be additional specifiers which dictate the type of access. These include:

- `ACCESS = 'SEQUENTIAL'` (the default) or `'DIRECT'`
- `FORM = 'FORMATTED'` (the default) or `'UNFORMATTED'`
- `STATUS = 'UNKNOWN'` (the default), `'OLD'`, `'NEW'` or `'REPLACE'`
- `ERR =` *label*

For example,
```
  OPEN (12, FILE = 'mydata.txt', ACCESS = 'SEQUENTIAL', STATUS = 'OLD', ERR = 999)
```
will branch to the statement with label `999` if file `mydata.txt` isn't found.

The general form of the `CLOSE` statement is
```
   CLOSE ( [UNIT = ]unit[, STATUS = status] )
```
where *status* may be either `'KEEP'` (the default) or `'DELETE'`.

10.3 Formatted Output

Example. The following code fragment indicates how `I`, `F` and `E` *edit specifiers* display numbers in integer, fixed-point and floating-point formats. The layout is determined by the `FORMAT` statement at label 100.

```
WRITE ( *, 100 ) 55, 55.0, 55.0
100 FORMAT ( 1X, 'I, F and E format: ', I3, 1X, F5.2, 1X, E8.2 )
```

This outputs (to the screen) the line
```
I, F and E formats:  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:

I*w*	integer in a field of width *w*;
F*w.d*	real, fixed-point format, in a field of width *w* with *d* decimal places;
E*w.d*	real, floating-point (*exponential*) format in a field of width *w* with *d* decimal places;
*n*PE*w.d*	floating point format as above with *n* significant figures in front of the decimal point;
L*w*	logical value (T or F) in a field of width *w*;
A*w*	character string in a field of width *w*;
A	character string of length determined by the output list;
'*text*'	a character string actually placed in the format specification;
*n*X	*n* spaces
T*n*	move to position *n* of the current record;
/	start a new record;

This is only a fraction of the available edit descriptors – see Hahn (1994).

For numerical output, if the required character representation is less than the specified width then it will be right-justified in the field. If the required number of characters exceeds the specified width then the field will be filled with asterisks. For example, an attempt to write 999 with edit descriptor `I2` will result in `**`.

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
```
If the whole format specifier isn't required (as in the last line of the above example) that doesn't matter: 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 above code example could also be written

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

Alternative Formatting Methods

The following are all equivalent means of specifying the same output format.

```
WRITE ( *, 150 ) X
150 FORMAT ( 1X, F5.2 )
```

```
WRITE ( *, '( 1X, F5.2 )' ) X
```

```
CHARACTER (LEN=*), C = '( 1X, F5.2 )'
WRITE ( *, C ) X
```

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

The general form of the `READ` statement is

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

unit and *format* are as for the corresponding `WRITE` statement. However, *format* is seldom anything other than `*` (i.e. list-directed input) with input data separated by blank spaces.

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. (Salford Fortran: –1 means EOF and –2 means EOR.)

10.5 File Positioning

Non-Advancing Output

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

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

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 previous record.

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

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.

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE or MODULE statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td rowspan="3">FORMAT
statements</td><td colspan="2">IMPLICIT NONE statement</td></tr>
<tr><td>PARAMETER and
DATA statements</td><td>type declarations</td></tr>
<tr><td colspan="2">executable statements</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

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

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE or MODULE statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td rowspan="3">FORMAT
statements</td><td colspan="2">IMPLICIT NONE statement</td></tr>
<tr><td>PARAMETER and
DATA statements</td><td>type declarations</td></tr>
<tr><td colspan="2">executable statements</td></tr>
<tr><td colspan="3">CONTAINS</td></tr>
<tr><td colspan="3">internal subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

A2. Fortran Statements

The following list is of the more common statements and is not exhaustive. A more complete list may be found in, e.g., Hahn (1994). 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 declared `KIND`.

A4. Intrinsic Routines

A comprehensive list can be found in Hahn, 1994.

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 and logarithmic 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 2×2 matrix
`MAXVAL(` *arra y*`)`, `MINVAL(` *array* `)` – maximum and minimum values (integer or real)
`PRODUCT(` *arra y*`)` – product of values (integer, real or complex)
`SUM(` *arra y*`)` – sum of values (integer, real or complex)

A5. Operators

Numeric Intrinsic Operators

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

Relational Operators

Operator	*Operation*
< 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

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

Character Operators

`//` concatenation