1. Character handling
2. Subroutines and functions

---

1. Character Handling

Example 1.1 Basic character-handling operations.

```
PROGRAM CHARACTER_FUNCTIONS
! Program illustrating strings and character functions
   IMPLICIT NONE
   CHARACTER (LEN=72) SCHOOL

   SCHOOL = 'School of Mechanical, Aerospace and Civil Engineering'

   PRINT *, '1: ', SCHOOL
   PRINT *, '2: ', SCHOOL(11:20)
   PRINT *, '3: ', SCHOOL(37:)
   PRINT *, '4: ', LEN( SCHOOL ), LEN_TRIM( SCHOOL ), LEN( TRIM( SCHOOL ) )
   PRINT *, '5: ', INDEX( SCHOOL, 'Civil' )
   PRINT *, '6: ', SCAN( SCHOOL, 'PQR' ), SCAN( SCHOOL, 'pqr' )
   PRINT *, '7: ', SCAN( SCHOOL, 'e' ), SCAN( SCHOOL, 'e', .TRUE. )
   PRINT *, '8: ', VERIFY( SCHOOL, 'Superb scholars' )

END PROGRAM CHARACTER_FUNCTIONS
```

Example 1.2 The Fortran character set.

```
PROGRAM CHARACTER_SET
! Program prints (most of) the Fortran character set, 10 to a line
   IMPLICIT NONE
   INTEGER N
   CHARACTER (LEN=*), PARAMETER :: FMT = '( 10( 2X, I3, 1X, A1 ) )'

   WRITE( *, FMT ) ( N, CHAR(N), N = 20, 127 )

END PROGRAM CHARACTER_SET
```

Example 1.3 Date, time and character variables.

```
PROGRAM CLOCK
! Program asking the computer for date and time
   IMPLICIT NONE
   CHARACTER (LEN=8) DATE            ! date in format ccyymmdd
   CHARACTER (LEN=10) TIME           ! time in format hhmmss.sss
   CHARACTER (LEN=5) ZONE            ! time zone (rel to UTC) as Shhmm
   INTEGER VALUES(8)                 ! year, month, day, mins from UTC,
                                     !      hours, min, sec, msec

   CHARACTER (LEN=8) TIMESTRING      ! time in digital form
   CHARACTER (LEN=20) DATESTRING     ! date in various forms
   CHARACTER (LEN=9) MONTH(12)       ! months of the year
   INTEGER FIRST, LAST               ! system clock counts
   INTEGER COUNTS_PER_SECOND         ! clock counts per second
   REAL SECONDS                      ! time taken

   DATA MONTH / 'January', 'February', 'March', 'April', 'May', 'June', &
                'July', 'August', 'September', 'October', 'November', 'December' /


   ! Ask for date and time (DATE and TIME as text, VALUES as integers)
   CALL DATE_AND_TIME( DATE, TIME, ZONE, VALUES )

   ! Put the time in digital form
   TIMESTRING = TIME( 1:2 ) // ':' // TIME( 3:4 ) // ':' // TIME( 5:6 )

   ! Date in abbreviated form from character string DATE
   DATESTRING = DATE( 7:8 ) // '-' // DATE( 5:6 ) // '-' // DATE( 1:4 )
   WRITE( *, * ) 'It is ', TIMESTRING, ' on ', DATESTRING

   ! Date from integer array VALUES
   WRITE( DATESTRING, '(I2, 1X, A, 1X, I4)') VALUES(3), TRIM(MONTH(VALUES(2))), VALUES(1)
   WRITE( *, * ) 'It is ', TIMESTRING, ' on ', DATESTRING


   ! Illustrates timing
   CALL SYSTEM_CLOCK( FIRST, COUNTS_PER_SECOND )
   WRITE( *, * ) 'Hit the enter button'
   READ( *, * )
   CALL SYSTEM_CLOCK( LAST, COUNTS_PER_SECOND )
   SECONDS = REAL( LAST - FIRST ) / COUNTS_PER_SECOND
   WRITE( *, * ) 'It took ', SECONDS, ' seconds to run this stupid program'

END PROGRAM CLOCK
```

2. Subprograms (Subroutines and Functions)

Example 2.1 Use of functions to allow *general* methods to be applied to *particular* problems.

(a) Trapezium rule for integration:

$$\int_a^b f(x)\ \mathrm{d}x \approx \frac{\Delta x}{2}\left[f(a) + f(b) + 2\sum_{i=1}^{N-1} f(x_i) \right]$$

where, with N intervals,

$$\Delta x = \frac{b-a}{N}, \qquad x_i = a + i\Delta x$$

```
PROGRAM TRAPEZIUM_RULE
   IMPLICIT NONE
   REAL, EXTERNAL :: F                        ! function to be integrated
   INTEGER N                                  ! number of intervals
   REAL A, B                                  ! limits of integration
   REAL INTEGRAL                              ! value of integral
   REAL DX                                    ! interval
   REAL X                                     ! an ordinate
   INTEGER I                                  ! a counter

   PRINT *, 'Input A, B, N'
   READ *, A, B, N

   DX = (B - A) / N                           ! calculate interval

   INTEGRAL = F(A) + F(B)                     ! contribution from ends
   DO I = 1, N - 1
      X = A + I * DX                          ! calculate intermediate point
      INTEGRAL = INTEGRAL + 2.0 * F(X)        ! add contribution to sum
   END DO

   INTEGRAL = INTEGRAL * DX / 2.0             ! convert sum to integral

   PRINT *, 'Integral = ', INTEGRAL

END PROGRAM TRAPEZIUM_RULE

!=========================================================================

REAL FUNCTION F( X )                          ! Function to be integrated
   IMPLICIT NONE
   REAL X

   F = X ** 2

END FUNCTION F
```

To change the function to be integrated just change the `F = …` line.

(b) Newton-Raphson iteration to solve $f(x) = 0$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Program the method to work for arbitrary *f*; use functions to return particular *f* and *f* '.

```
PROGRAM NEWTON_RAPHSON
! Program finds a root of the equation f(x) = 0 by the Newton-Raphson method
   IMPLICIT NONE
   REAL, EXTERNAL :: F, DFDX                 ! function and its derivative
   REAL, PARAMETER :: TOLERANCE = 1.0E-6     ! tolerance for near-zero
   INTEGER, PARAMETER :: ITERMX = 200        ! maximum number of iterations
   REAL X                                    ! current x
   REAL FVALUE                               ! current value of function f
   INTEGER :: ITER = 0                       ! current iteration number

   PRINT *, 'Input initial X'
   READ *, X                                 ! input first guess
   FVALUE = F( X )                           ! value of function
   PRINT *, 'X, F(X) =', X, FVALUE

   ! Loop until root found or maximum iterations reached
   DO WHILE ( ABS( FVALUE ) > TOLERANCE .AND. ITER <= ITERMX )
      X = X - FVALUE / DFDX( X )             ! update x by Newton-Raphson formula
      FVALUE = F( X )                        ! update value of function
      ITER = ITER + 1                        ! update iteration number
      PRINT *, 'X, F(X) =', X, FVALUE        ! output current values
   END DO

   ! Output answer (or warn if not converged)
   IF ( ABS( FVALUE ) > TOLERANCE ) THEN
      PRINT *, 'Not converged'
   ELSE
      PRINT *, 'Answer =', X
   END IF

END PROGRAM NEWTON_RAPHSON

!=========================================

REAL FUNCTION F( X )
! This function should return the value of the function at X
   IMPLICIT NONE
   REAL X

   F = 16.0 * X * X - 4.0

END FUNCTION F

!=========================================

REAL FUNCTION DFDX( X )
! This function should return the derivative of the function at X
   IMPLICIT NONE
   REAL X

   DFDX = 32.0 * X

END FUNCTION DFDX
```

To change the equation to be solved just change the `F = …` and `DFDX = …` lines.

Example 2.2 Use of subroutines.

```
PROGRAM EXAMPLE
! Program to swap two numbers
   IMPLICIT NONE
   EXTERNAL SWAP                  ! (optionally) declare routine to be used
   INTEGER I, J

   PRINT *, 'Input integers I and J'
   READ *, I, J

   CALL SWAP( I, J )

   PRINT *, 'Swapped numbers are ', I, J

END PROGRAM EXAMPLE

!=======================================

SUBROUTINE SWAP( M, N )
   IMPLICIT NONE
   INTEGER M, N                   ! numbers to be swapped
   INTEGER TEMP                   ! temporary storage

   TEMP = M                       ! store number before changing it
   M = N
   N = TEMP

END SUBROUTINE SWAP
```

Example 2.3 Specifying `INTENT` for subprogram arguments.

```
PROGRAM COORDINATES
! Program to convert from Cartesian to polar coordinates
   IMPLICIT NONE
   EXTERNAL POLARS
   REAL X, Y
   REAL R, THETA

   PRINT *, 'Input coordinates X and Y'
   READ *, X, Y

   CALL POLARS( X, Y, R, THETA )
   PRINT *, 'R, THETA =', R, THETA

END PROGRAM COORDINATES

!========================

SUBROUTINE POLARS( X, Y, R, THETA )
! Subroutine transforming input (X, Y) to output (R, THETA)
   IMPLICIT NONE
   REAL, INTENT(IN) :: X, Y            ! cartesian coordinates (input)
   REAL, INTENT(OUT) :: R, THETA       ! polar coordinates (output)
   REAL, PARAMETER :: PI = 3.141593    ! the constant pi

   R = SQRT( X ** 2 + Y ** 2 )         ! radius

   THETA = ATAN2( Y, X )               ! inverse tangent between -pi and pi
   IF ( THETA < 0.0 ) THETA = THETA + 2.0 * PI
                                       ! angle between 0 and 2 pi
   THETA = THETA * 180.0 / PI          ! convert to degrees

END SUBROUTINE POLARS
```

Example 2.4 Subroutines with array arguments.

Mean: $\overline{X} = \frac{\sum X}{N}$, sample variance: $\sigma^2 = \frac{\sum X^2}{N} - \overline{X}^2$, standard deviation: σ

(*Note*: for an unbiased estimate of the population variance multiply σ^2 by $\frac{N}{N-1}$.)

```
PROGRAM EXAMPLE
! Program computes mean, variance and standard deviation
   IMPLICIT NONE
   EXTERNAL STATS                                  ! subroutine to be used
   INTEGER NVAL                                    ! number of values
   REAL, ALLOCATABLE :: X(:)                       ! data values
   REAL MEAN, VARIANCE, STANDARD_DEVIATION         ! statistics
   INTEGER N                                       ! a counter

   ! Open data file
   OPEN( 10, FILE = 'stats.dat' )

   ! Read the number of points and set aside enough memory
   READ( 10, * ) NVAL
   ALLOCATE( X(NVAL) )

   ! Read data values
   READ( 10, * ) ( X(N), N = 1, NVAL )
   CLOSE( 10 )

   ! Compute statistics
   CALL STATS( NVAL, X, MEAN, VARIANCE, STANDARD_DEVIATION )

   ! Output results
   PRINT *, 'Mean = ', MEAN
   PRINT *, 'Variance = ', VARIANCE
   PRINT *, 'Standard deviation = ', STANDARD_DEVIATION

   ! Recover computer memory
   DEALLOCATE( X )

END PROGRAM EXAMPLE

!=================================================

SUBROUTINE STATS( N, X, M, VAR, SD )
! This works out the sample mean, variance and standard deviation
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: N           ! array size
   REAL, INTENT(IN) :: X(N)           ! data values
   REAL, INTENT(OUT) :: M, VAR, SD    ! statistics

   ! Calculate statistics using array operation SUM
   M = SUM( X ) / N                   ! mean
   VAR = SUM( X * X ) / N - M ** 2    ! variance
   SD = SQRT( VAR )                   ! standard deviation

END SUBROUTINE STATS
```

1. Input and output
2. Modules

---

1. Input and Output

Example 1.1 Illustrating formatted output.

```
PROGRAM TRIG_TABLE
! Compiles a table of SIN, COS, TAN against angle in DEGREES
   IMPLICIT NONE
   INTEGER DEG                          ! angle in degrees
   REAL RAD                             ! angle in radians
   REAL PI                              ! mathematical pi
   CHARACTER (LEN=*), PARAMETER :: FMTHEAD = '( 1X, A3, 3( 2X, A7   ) )'
   CHARACTER (LEN=*), PARAMETER :: FMTDATA = '( 1X, I3, 3( 2X, F7.4 ) )'
                                        ! formats for headings and data

   PI = 4.0 * ATAN( 1.0 )

   WRITE( *, FMTHEAD ) 'Deg', 'Sin', 'Cos', 'Tan'
   DO DEG = 0, 80, 10
      RAD = DEG * PI / 180.0
      WRITE( *, FMTDATA ) DEG, SIN( RAD ), COS( RAD ), TAN( RAD )
   END DO

END PROGRAM TRIG_TABLE
```

Alternatively:

```
   WRITE( *,'( 1X, A3, 3( 2X, A7 ) )' ) 'Deg', 'Sin', 'Cos', 'Tan'
   …
      WRITE( *,'( 1X, I3,3( 2X, F7.4 ) )' ) DEG, SIN( RAD ), COS( RAD ),TAN( RAD )
```

or:

```
   100 FORMAT( 1X, A3, 3( 2X, A7   ) )
   110 FORMAT( 1X, I3, 3( 2X, F7.4 ) )
   …
   WRITE( *, 100 ) 'Deg', 'Sin', 'Cos', 'Tan'
   …
      WRITE( *, 110 ) DEG, SIN( RAD ), COS( RAD ), TAN( RAD )
   …
```

Writing to file:

```
   OPEN( 33, FILE = 'trig.out' )      ! open file for output
   WRITE( 33, FMTHEAD ) 'Deg', 'Sin', 'Cos', 'Tan'
   …
      WRITE( 33, FMTDATA ) DEG, SIN( RAD ), COS( RAD ), TAN( RAD )
   …
   CLOSE( 33 )                        ! close file (tidiness is a virtue!)
```

Example 1.2 Illustrating floating-point output.

```
PROGRAM EXPTABLE
! Program tabulates EXP(X)
    IMPLICIT NONE
    INTEGER :: NSTEP = 15                          ! number of steps
    REAL :: XMIN = 0.0, XMAX = 3.0                 ! interval limits
    REAL DELTAX                                    ! step size
    REAL X                                         ! current X value
    INTEGER I                                      ! a counter

    ! Format specifiers
    CHARACTER (LEN=*), PARAMETER :: FMT1 = '( 1X, A4  , 2X,   A10   )'
    CHARACTER (LEN=*), PARAMETER :: FMT2 = '( 1X, F4.2, 2X, 1PE10.3 )'

    DELTAX = ( XMAX - XMIN ) / NSTEP               ! calculate step size
    WRITE( *, FMT1 ) 'X', 'EXP'                    ! write headers

    DO I = 0, NSTEP
       X = XMIN + I * DELTAX                       ! set X value
       WRITE( *, FMT2 ) X, EXP( X )                ! write data
    END DO

END PROGRAM EXPTABLE
```

Example 1.3 Illustrating reading and writing files and data analysis.

The program adds scores from a round of golf.

It reads an input file `scorecard.txt` of the form

```
4
6
        Hole: 1    2    3    4    5    6
David         5    5    11   3    3    7
Judith        3    6    5    5    1    4
Miriam        4    3    9    2    4    6
Elizabeth     4    4    5    3    2    4
```

where:

- the first line is the number of players (here, 4)
- the second line is the number of holes (here, 6)
- the third line is the set of hole numbers
- the remaining lines are the players' names and their scores for each hole.

The number of players and the number of holes is arbitrary.

Output consists of a table of players and total scores and is written to file `total.txt`.

```
PROGRAM GOLF
   IMPLICIT NONE
   INTEGER NPLAYERS                                ! number of files
   INTEGER NHOLES                                  ! number of holes
   INTEGER SCORE                                   ! total score
   INTEGER I                                       ! loop counter
   INTEGER, ALLOCATABLE :: SHOTS(:)                ! score for each hole
   CHARACTER (LEN=15) PLAYER                       ! player name
   CHARACTER (LEN=*), PARAMETER :: FMT = '(A, 3X, I3)'   ! output format

   ! Open files
   OPEN( 21, FILE='scorecard.txt' )
   OPEN( 22, FILE='total.txt' )

   ! Read numbers of players and holes
   READ( 21, * ) NPLAYERS
   READ( 21, * ) NHOLES
   READ( 21, * )                        ! skip a line - nothing needed

   ! Allocate memory to record the scores of one player
   ALLOCATE( SHOTS(NHOLES) )

   ! Loop round, reading player and scores, writing player and total
   DO I = 1, NPLAYERS
      READ( 21, * ) PLAYER, SHOTS
      SCORE = SUM( SHOTS )
      WRITE( 22, FMT ) PLAYER, SCORE
   END DO

   ! Close files and tidy up
   CLOSE( 21 )
   CLOSE( 22 )
   DEALLOCATE( SHOTS )

END PROGRAM GOLF
```

Example 1.4 Illustrating formatted `READ`, non-advancing i/o and the `IOSTAT` specifier.

The program converts a passage of text to upper case.

Input file: `input.txt` (any favourite piece of literature will do!)
Output file: `output.txt`

```
PROGRAM UPPERCASE
   IMPLICIT NONE
   INTEGER :: IO = 0
   INTEGER :: INCREMENT = ICHAR( 'A' ) - ICHAR( 'a' )
   CHARACTER CH

   OPEN( 10, FILE = 'input.txt' )                    ! Open files
   OPEN( 20, FILE = 'output.txt' )

   DO WHILE ( IO /= -1 )
      READ( 10, '( A1 )', IOSTAT = IO, ADVANCE = 'NO' ) CH

      IF ( IO == 0 ) THEN
         IF ( CH >= 'a' .AND. CH <= 'z' ) THEN      ! Change to upper case
            CH = CHAR( ICHAR( CH ) + INCREMENT )
         END IF
         WRITE (20, '( A1 )', ADVANCE = 'NO' ) CH   ! Write to file
      ELSE IF ( IO == -2 ) THEN
         WRITE (20, *)                              ! Force a line feed
      END IF

   END DO

   CLOSE( 10 )                                      ! Close files
   CLOSE( 20 )

END PROGRAM UPPERCASE
```

2. Modules

Example 2.1 Illustrating modules used to share related parameters and variables.

Compilation and linking commands:

```
ftn95 conversion.f95
ftn95 distance.f95
slink distance.obj conversion.obj
```

conversion.f95

```
MODULE CONVERSION
! Length conversion factors
   IMPLICIT NONE
   REAL, PARAMETER :: MILES_TO_METRES  = 1609.0
   REAL, PARAMETER :: YARDS_TO_METRES  = 0.9144
   REAL, PARAMETER :: FEET_TO_METRES   = 0.3048
   REAL, PARAMETER :: INCHES_TO_METRES = 0.0254

END MODULE CONVERSION
```

distance.f95

```
PROGRAM DISTANCE
   USE CONVERSION           ! make conversion factors available

   IMPLICIT NONE
   REAL METRES              ! distance in metres
   REAL AMOUNT              ! numerical quantity
   CHARACTER UNITS          ! units (i-inches, f-feet, y-yards, m-miles)

   PRINT *, 'Input amount and units (i-inches, f-feet, y-yard, m-mile)'
   READ *, AMOUNT, UNITS

   SELECT CASE (UNITS)
      CASE ( 'i' , 'I' );  METRES = AMOUNT * INCHES_TO_METRES
      CASE ( 'f' , 'F' );  METRES = AMOUNT * FEET_TO_METRES
      CASE ( 'y' , 'Y' );  METRES = AMOUNT * YARDS_TO_METRES
      CASE ( 'm' , 'M' );  METRES = AMOUNT * MILES_TO_METRES
   END SELECT

   PRINT *, 'Distance in metres = ', METRES

END PROGRAM DISTANCE
```

Example 2.2 Illustrating modules as `CONTAIN`ers of useful routines.

Compilation and linking commands:

```
ftn95 physics.f95
ftn95 gas.f95
slink gas.obj physics.obj
```

`physics.f95`

```
MODULE PHYSICS
! Physical constants and some useful subprograms
   IMPLICIT NONE
   REAL, PARAMETER :: SPEED_OF_LIGHT            = 3.00E+08     ! (m/s)
   REAL, PARAMETER :: PLANCKS_CONSTANT          = 6.63E-34     ! (J s)
   REAL, PARAMETER :: GRAVITATIONAL_CONSTANT    = 6.67E-11     ! (N m2/kg2)
   REAL, PARAMETER :: ELECTRON_MASS             = 9.11E-31     ! (kg)
   REAL, PARAMETER :: ELECTRON_CHARGE           = 1.60E-19     ! (C)
   REAL, PARAMETER :: STEFAN_BOLTZMANN_CONSTANT = 5.67E-08     ! (W/m2/K4)
   REAL, PARAMETER :: IDEAL_GAS_CONSTANT        = 8.31E+00     ! (J/K)
   REAL, PARAMETER :: AVOGADRO_NUMBER           = 6.02E+23     ! (/mol)

   CONTAINS

      REAL FUNCTION PRESSURE( n, T, V )
      ! Computes pressure by ideal gas law (pV=nRT)
         REAL n, T, V                     ! moles, temperature, volume

         PRESSURE = n * IDEAL_GAS_CONSTANT * T / V
      END FUNCTION PRESSURE

      REAL FUNCTION RADIATION( T, AREA, EMISSIVITY )
      ! Computes radiative heat flux
         REAL T                           ! thermodynamic temperature
         REAL AREA                        ! area of surface
         REAL EMISSIVITY

         RADIATION = EMISSIVITY * STEFAN_BOLTZMANN_CONSTANT * T ** 4 * AREA
      END FUNCTION RADIATION

END MODULE PHYSICS
```

`gas.f95`

```
PROGRAM IDEAL_GAS
! Program to test module <physics>
   USE PHYSICS                      ! access module

   IMPLICIT NONE
   REAL RMM                         ! relative molecular mass
   REAL MASS                        ! mass (kg)
   REAL TEMPERATURE                 ! temperature (K)
   REAL VOLUME                      ! volume (m3)
   REAL MOLES                       ! moles of gas

   PRINT *, 'Input relative molecular mass'
   READ *, RMM
   PRINT *, 'Input mass(kg), temperature(K), volume(m3)'
   READ *, MASS, TEMPERATURE, VOLUME

   MOLES = 1000.0 * MASS / RMM      ! calculate moles of gas

   PRINT *, 'Pressure = ', PRESSURE( MOLES, TEMPERATURE, VOLUME ), 'Pa'

END PROGRAM IDEAL_GAS
```

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

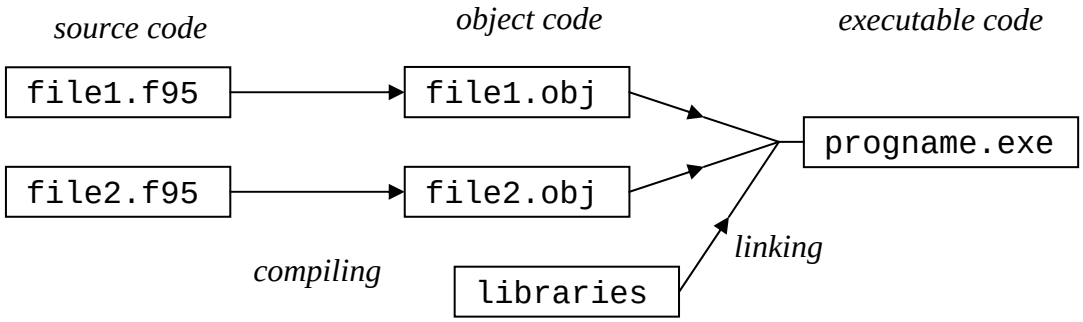
The compiler which we shall use is provided by Salford Software and happens to be a Fortran 95 compiler. However, everything we do will actually conform to the Fortran 90 standard.

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

Answers to Selected Exercises for

Fortran 90/95 for Scientists and Engineers

by Stephen J. Chapman

Chapter 1. Introduction to Computers and the Fortran Language

1-1 *(a)* 1010_2 *(c)* 1001101_2

1-2 *(a)* 72_{10} *(c)* 255_{10}

1-5 A 23-bit mantissa can represent approximately $\pm 2^{22}$ numbers, or about six significant decimal digits. A 9-bit exponent can represent multipliers between 2^{-255} and 2^{255}, so the range is from about 10^{-76} to 10^{76}.

1-8 The sum of the two's complement numbers is:

0010010010010010_2 = 9362_{10}

$\underline{1111110011111100}_2$ = $\underline{-772}_{10}$

0010000110001110_2 = 8590_{10}

The two answers agree with each other.

Chapter 2. Basic Elements of Fortran

2-1 *(a)* Valid real constant *(c)* Invalid constant—numbers may not include commas *(e)* Invalid constant—need two single quotes to represent a quote within a string

2-4 *(a)* Legal: result = 0.888889 *(c)* Illegal—cannot raise a negative real number to a negative real power

2-12 The output of the program is:

```
      -3.141592      100.000000      200.000000         300        -100        -200
```

2-13 The weekly pay program is shown below:

```
PROGRAM get_pay
!
!  Purpose:
!    To calculate an hourly employee's weekly pay.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    12/30/96    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
```

```
REAL :: hours        ! Number of hours worked in a week.
REAL :: pay          ! Total weekly pay.
REAL :: pay_rate     ! Pay rate in dollars per hour.

! Get pay rate
WRITE (*,*) 'Enter employees pay rate in dollars per hour: '
READ (*,*) pay_rate

! Get hours worked
WRITE (*,*) 'Enter number of hours worked: '
READ (*,*) hours

! Calculate pay and tell user.
pay = pay_rate * hours
WRITE (*,*) "Employee's pay is $", pay

END PROGRAM
```

The result of executing this program is

```
C:\BOOK\F90\SOLN>get_pay
Enter employees pay rate in dollars per hour:
5.50
Enter number of hours worked:
39
Employee's pay is $     214.500000
```

2-17 A program to calculate the hypotenuse of a triangle from the two sides is shown below:

```
PROGRAM calc_hypotenuse
!
!  Purpose:
!    To calculate the hypotenuse of a right triangle, given
!    the lengths of its two sides.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    12/30/96    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
REAL :: hypotenuse   ! Hypotenuse of triangle
REAL :: side_1       ! Side 1 of triangle
REAL :: side_2       ! Side 2 of triangle

! Get lengths of sides.
WRITE (*,*) 'Program to calculate the hypotenuse of a right '
WRITE (*,*) 'triangle, given the lengths of its sides. '
WRITE (*,*) 'Enter the length side 1 of the right triangle: '
READ (*,*) side_1
WRITE (*,*) 'Enter the length side 2 of the right triangle: '
READ (*,*) side_2

! Calculate length of the hypotenuse.
```

```
hypotenuse = SQRT ( side_1**2 + side_2**2 )

! Write out results.
WRITE (*,*) 'The length of the hypotenuse is: ', hypotenuse

END PROGRAM
```

2-21 A program to calculate the hyperbolic cosine is shown below:

```
PROGRAM coshx
!
!  Purpose:
!    To calculate the hyperbolic cosine of a number.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    12/30/96    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
REAL :: result             ! COSH(x)
REAL :: x                  ! Input value

WRITE (*,*) 'Enter number to calculate cosh() of: '
READ (*,*) x

result = ( EXP(x) + EXP(-x) ) / 2.

WRITE (*,*) 'COSH(X) =', result

END PROGRAM
```

When this program is run, the result is:

```
C:\BOOK\F90\SOLN>coshx
Enter number to calculate cosh() of:
3.0
COSH(X) =      10.067660
```

The Fortran 90 intrinsic function `COSH()` produces the same answer.

Chapter 3. Control Structures and Program Design

3-2 The statements to calculate *y(t)* for values of *t* between -9 and 9 in steps of 3 are:

```
IMPLICIT NONE
INTEGER :: i
REAL :: t, y

DO i = -9, 9, 3
   t = REAL(i)
```

```
   IF ( t >= 0. ) THEN
      y = -3.0 * t**2 + 5.0
   ELSE
      y = 3.0 * t**2 + 5.0
   END IF
   WRITE (*,*) 't = ', t, '   y(t) = ', y
END DO
END PROGRAM
```

3-6 The statements are incorrect. In an `IF` construct, the first branch whose condition is true is executed, and all others are skipped. Therefore, if the temperature is 104.0, then the second branch would be executed, and the code would print out `'Temperature normal'` instead of `'Temperature dangerously high'`. A correct version of the `IF` construct is shown below:

```
IF ( temp < 97.5 ) THEN
   WRITE (*,*) 'Temperature below normal'
ELSE IF ( TEMP > 103.0 ) THEN
   WRITE (*,*) 'Temperature dangerously high'
ELSE IF ( TEMP > 99.5 ) THEN
   WRITE (*,*) 'Temperature slightly high'
ELSE IF ( TEMP > 97.5 ) THEN
   WRITE (*,*) 'Temperature normal'
END IF
```

3-14 *(a)* This loop is executed 21 times, and afterwards `ires` = 21. *(c)* This outer loop is executed 4 times, the inner loop is executed 13 times, and afterwards `ires` = 42.

3-19 The legal values of x for this function are all $x < 1.0$, so the program should contain a while loop which calculates the function $y(x) = \ln \frac{1}{1-x}$ for any x < 1.0, and terminates when $x \geq 1.0$ is entered.

```
PROGRAM evaluate
!
!  Purpose:
!    To evaluate the function ln(1./(1.-x)).
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!     12/30/96    S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare local variables:
REAL :: value      ! Value of function ln(1./(1.-x))
REAL :: x          ! Independent variable

! Loop over all valid values of x
DO

   ! Get next value of x.
   WRITE (*,*) 'Enter value of x: '
   READ (*,*) x

   ! Check for invalid value
```

```
   IF ( x >= 1. ) EXIT

   ! Calculate and display function
   value = LOG ( 1. / ( 1. - x ) )
   WRITE (*,*) 'LN(1./(1.-x)) = ', value

END DO

END PROGRAM
```

3-28 A program to calculate the harmonic of an input data set is shown below. This problem gave the student the freedom to input the data in any way desired; I have chosen a `DO` loop for this example program.

```
PROGRAM harmon
!
!  Purpose:
!    To calculate harmonic mean of an input data set, where each
!    input value can be positive, negative, or zero.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    12/31/96    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
REAL :: h_mean          ! Harmonic mean
INTEGER :: i            ! Loop index
INTEGER :: n            ! Number of input samples
REAL :: sum_rx = 0.     ! Sum of reciprocals of input values
REAL :: x = 0.          ! Input value

! Get the number of points to input.
WRITE (*,*) 'Enter number of points: '
READ  (*,*) n

! Loop to read input values.
DO i = 1, n

   ! Get next number.
   WRITE (*,*) 'Enter next number: '
   READ  (*,*) x

   ! Accumulate sums.
   sum_rx = sum_rx + 1.0 / x

END DO

! Calculate the harmonic mean
h_mean = REAL (n) / sum_rx

! Tell user.
WRITE (*,*) 'The harmonic mean of this data set is:', h_mean
WRITE (*,*) 'The number of data points is:         ', n
```

```
END PROGRAM
```

When the program is run with the sample data set, the results are:

```
C:\BOOK\F90\SOLN>harmon
Enter number of points:
4
Enter next number:
10.
Enter next number:
5.
Enter next number:
2.
Enter next number:
5.
The harmonic mean of this data set is:       4.000000
The number of data points is:                   4
```

Chapter 4. Basic I/O Concepts

4-3 *(b)* The result is printed out on the next line. It is:

```
A =   1.002000E+06 B =    .100010E+07 Sum =    .200210E+07 Diff =    1900.000000
---------|---------|---------|---------|---------|---------|---------|---------|
         10        20        30        40        50        60        70        80
```

4-8 A program to calculate the average and standard deviation of an input data set stored in a file is shown below:

```
PROGRAM ave_sd
!
!    To calculate the average (arithmetic mean) and standard
!    deviation of an input data set found in a user-specified
!    file, with the data arranged so that there is one value
!    per line.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/04/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare variables
REAL :: ave                         ! Average (arithmetic mean)
CHARACTER(len=20) :: filename       ! Name of file to open
INTEGER :: nvals = 0                ! Number of values read in
REAL :: sd                          ! Standard deviation
INTEGER :: status                   ! I/O status
REAL :: sum_x = 0.0                 ! Sum of the input values
REAL :: sum_x2 = 0.0                ! Sum of input values squared
REAL :: value                       ! The real value read in
```

```
! Get the file name, and echo it back to the user.
WRITE (*,1000)
1000 FORMAT (1X,'This program calculates the average and standard ' &
          ,/,1X,'deviation of an input data set.  Enter the name' &
          ,/,1X,'of the file containing the input data:' )
READ  (*,*) filename

! Open the file, and check for errors on open.
OPEN (UNIT=3, FILE=filename, STATUS='OLD', ACTION='READ', &
      IOSTAT=status )
openif: IF ( status == 0 ) THEN

   ! OPEN was ok.  Read values.
   readloop: DO
      READ (3,*,IOSTAT=status) value    ! Get next value
      IF ( status /= 0 ) EXIT           ! EXIT if not valid.
      nvals = nvals + 1                 ! Valid: increase count
         sum_x = sum_x + value             ! Sums
         sum_x2 = sum_x2 + value**2        ! Sum of squares
   END DO readloop

   ! The WHILE loop has terminated.  Was it because of a READ
   ! error or because of the end of the input file?
   readif: IF ( status > 0 ) THEN ! a READ error occurred.  Tell user.

      WRITE (*,1020) nvals + 1
      1020 FORMAT ('0','An error occurred reading line ', I6)

   ELSE ! the end of the data was reached.  Calculate ave & sd.

      ave = sum_x / REAL(nvals)
      sd  = SQRT( (REAL(nvals)*sum_x2-sum_x**2)/(REAL(nvals)*REAL(nvals-1)))
      WRITE (*,1030) filename, ave, sd, nvals
      1030 FORMAT ('0','Statistical information about data in file ',A,  &
                  /,1X,'  Average            = ', F9.3,     &
                  /,1X,'  Standard Deviation = ', F9.3,     &
                  /,1X,'  No of points       = ', I9 )
   END IF readif

ELSE openif
   WRITE (*,1040) status
   1040 FORMAT (' ','Error opening file: IOSTAT = ', I6 )
END IF openif

! Close file
CLOSE ( UNIT=8 )

END PROGRAM
```

Chapter 5. Arrays

5-4 *(a)* 60 elements; valid subscript range is 1 to 60. *(c)* 105 elements; valid subscript range is (1,1) to (35,3).

5-5 *(a)* Valid. These statements declare and initialize the 100-element array `icount` to 1, 2, 3, …, 100, and the 100-element array `jcount` to 2, 3, 4, …, 101. *(d)* Valid. The `WHERE` construct multiplies the positive elements of array `info` by -1, and negative elements by -3. It then writes out the values of the array: -1, 9, 0, 15, 27, -3, 0, -1, -7.

$$z = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

5-9 *(b)* The `READ` statement here reads all values from the first line, then all the values from the second line, etc. until 16 values have been read. The values are stored in array `values` in row order. Therefore, array `values` will contain the following values

$$\texttt{values} = \begin{bmatrix} 27 & 17 & 10 & 8 \\ 6 & 11 & 13 & -11 \\ 12 & -21 & -1 & 0 \\ 0 & 6 & 14 & -16 \end{bmatrix}$$

5-25 A program to calculate the distance between two points (x_1, y_1, z_1) and (x_2, y_2, z_2) in three-dimensional space is shown below:

```
PROGRAM dist_3d
!
!  Purpose:
!    To calculate the distance between two points (x1,y1,z1)
!    and (x2,y2,z2) in three-dimensional space.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/05/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of variables:
REAL :: dist       ! Distance between the two points.
REAL :: x1         ! x-component of first vector.
REAL :: x2         ! x-component of second vector.
REAL :: y1         ! y-component of first vector.
REAL :: y2         ! y-component of second vector.
REAL :: z1         ! z-component of first vector.
REAL :: z2         ! z-component of second vector.

! Get the first point in 3D space.
WRITE (*,1000)
1000 FORMAT (' Calculate the distance between two points ',&
               '(X1,Y1,Z1) and (X2,Y2,Z2):'&
           /,1X,'Enter the first point (X1,Y1,Z1): ')
READ (*,*) x1, y1, z1

! Get the second point in 3D space.
WRITE (*,1010)
```

```
1010 FORMAT (' Enter the second point (X2,Y2,Z2): ')
      READ (*,*) x2, y2, z2

! Calculate the distance between the two points.
dist = SQRT ( (x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2 )

! Tell user.
WRITE (*,1020) dist
1020 FORMAT (' The distance between the two points is ', F10.3)

END PROGRAM
```

When this program is run with the specified data values, the results are

```
C:\BOOK\F90\SOLN>dist_3d
Calculate the distance between two points (X1,Y1,Z1) and (X2,Y2,Z2):
Enter the first point (X1,Y1,Z1):
-1. 4. 6.
Enter the second point (X2,Y2,Z2):
1. 5. -2.
The distance between the two points is      8.307
```

Chapter 6. Procedures and Structured Programming

6-1 A function is a procedure whose result is a single number, logical value, or character string, while a subroutine is a subprogram that can return one or more numbers, logical values, or character strings. A function is invoked by naming it in a Fortran expression, while a subroutine is invoked using the `CALL` statement.

6-6 Data is passed by reference from a calling program to a subroutine. Since only a pointer to the location of the data is passed, *there is no way for a subroutine with an implicit interface to know that the argument type is mismatched*. (However, some Fortran compilers are smart enough to recognize such type mismatches if both the calling program and the subroutine are contained in the same source file.)

The result of executing this program will vary from processor. When executed on a computer with IEEE standard floating-point numbers, the results are

```
C:\BOOK\F90\SOLN>main
 I = -1063256064
```

6-10 According to the Fortran 90/95 standard, the values of all the local variables in a procedure become undefined whenever we exit the procedure. The next time that the procedure is invoked, the values of the local variables may or may not be the same as they were the last time we left it, depending on the particular processor being used. If we write a procedure that depends on having its local variables undisturbed between calls, it will work fine on some computers and fail miserably on other ones!

Fortran provides the `SAVE` attribute and the `SAVE` statement to guarantee that local variables are saved unchanged between invocations of a procedure. Any local variables declared with the `SAVE` attribute or listed in a `SAVE` statement will be saved unchanged. If no variables are listed in a `SAVE` statement, then all of the local variables will be saved unchanged. In addition, any local variables that are initialized in a type declaration statement will be saved unchanged between invocations of the procedure.

6-26 A subroutine to calculate the derivative of a discrete function is shown below.

```
SUBROUTINE derivative ( vector, deriv, nsamp, dx, error )
!
!  Purpose:
!    To calculate the derivative of a sampled function f(x)
!    consisting of nsamp samples spaced a distance dx apart.
!    The resulting derivative is returned in array deriv, and
!    is nsamp-1 samples long.  (Since calculating the derivative
!    requires both point i and point i+1, we can't find the
!    derivative for the last point in the input array.)
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/07/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of calling arguments:
INTEGER,INTENT(IN) :: nsamp                  ! Number of samples
REAL,DIMENSION(nsamp),INTENT(IN) :: vector   ! Input data array
REAL,DIMENSION(nsamp-1),INTENT(OUT) :: deriv ! Input data array
REAL,INTENT(IN) :: dx                        ! sample spacing
INTEGER,INTENT(OUT) :: error                 ! Flag: 0 = no error
                                             !       1 = dx <= 0

! List of local variables:
INTEGER :: i                                 ! Loop index

! Check for legal step size.
IF ( dx > 0. ) THEN

   ! Calculate derivative.
   DO i = 1, nsamp-1
      deriv(i) = ( vector(i+1) - vector(i) ) / dx
   END DO
   error = 0

ELSE

   ! Illegal step size.
   error = 1

END IF

END SUBROUTINE
```

A test driver program for this subroutine is shown below. This program creates a discrete analytic function $f(x) = \sin x$, and calculates the derivative of that function using subroutine `DERV`. Finally, it compares the result of the subroutine to the analytical solution $df(x)/dx = \cos x$, and find the maximum difference between the result of the subroutine and the true solution.

```
PROGRAM test_derivative
!
!  Purpose:
!    To test subroutine "derivative", which calculates the numerical
```

```
!    derivative of a sampled function f(x).  This program will take the
!    derivative of the function f(x) = sin(x), where nstep = 100, and
!    dx = 0.05.  The program will compare the derivative with the known
!    correct answer df/dx = cox(x)), and determine the error in the
!    subroutine.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/07/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of named constants:
INTEGER,PARAMETER :: nsamp = 100      ! Number of samples
REAL,PARAMETER :: dx = 0.05           ! Step size

! List of local variables:
REAL,DIMENSION(nsamp-1) :: cderiv     ! Analytically calculated deriv
REAL,DIMENSION(nsamp-1) :: deriv      ! Derivative from subroutine
INTEGER :: error                      ! Error flag
INTEGER :: i                          ! Loop index
REAL :: max_error                     ! Max error in derivative
REAL,DIMENSION(nsamp) :: vector       ! f(x)

! Calculate f(x)
DO i = 1, nsamp
   vector(i) = SIN ( REAL(i-1) * dx )
END DO

! Calculate analytic derivative of f(x)
DO i = 1, nsamp-1
   cderiv(i) = COS ( REAL(i-1) * dx )
END DO

! Call "derivative"
CALL derivative ( vector, deriv, nsamp, dx, error )

! Find the largest difference between the analytical derivative and
! the result of subroutine "derivative".
max_error = MAXVAL ( ABS( deriv - cderiv ) )

! Tell user.
WRITE (*,1000) max_error
1000 FORMAT (' The maximum error in the derivative is ', F10.4,'.')

END PROGRAM
```

When this program is run, the results are

```
C:\BOOK\F90\SOLN>test_derivative
The maximum error in the derivative is      .0250.
```

7-4 A character function version of `ucase` is shown below. In order to return a variable-length character string, this function must have an explicit interface, so it is embedded in a module here.

```
MODULE myprocs

CONTAINS
   FUNCTION ucase ( string )
   !
   !  Purpose:
   !    To shift a character string to upper case on any processor,
   !    regardless of collating sequence.
   !
   !  Record of revisions:
   !      Date       Programmer          Description of change
   !      ====       ==========          =====================
   !    01/09/96    S. J. Chapman        Original code
   !
   IMPLICIT NONE

   ! Declare calling parameters:
   CHARACTER(len=*), INTENT(IN) :: string      ! Input string
   CHARACTER(len=LEN(string)) :: ucase         ! Function

   ! Declare local variables:
   INTEGER :: i                 ! Loop index
   INTEGER :: length            ! Length of input string

   ! Get length of string
   length = LEN ( string )

   ! Now shift lower case letters to upper case.
   DO i = 1, length
      IF ( LGE(string(i:i),'a') .AND. LLE(string(i:i),'z') ) THEN
         ucase(i:i) = ACHAR ( IACHAR ( string(i:i) ) - 32 )
      ELSE
         ucase(i:i) = string(i:i)
      END IF
   END DO

   END FUNCTION ucase

END MODULE
```

A simple test driver program is shown below.

```
PROGRAM test_ucase
!
!  Purpose:
!    To test function ucase.
!
!  Record of revisions:
!      Date       Programmer          Description of change
```

```
!      ====       ==========          ====================
!    01/09/96    S. J. Chapman        Original code
!
USE myprocs
IMPLICIT NONE
CHARACTER(len=30) string
WRITE (*,*) 'Enter test string (up to 30 characters): '
READ (*,'(A30)') string
WRITE (*,*) 'The shifted string is: ', ucase(string)
END PROGRAM
```

When this program is executed, the results are:

```
C:\BOOK\F90\SOLN>test_ucase
Enter test string (up to 30 characters):
This is a Test! 12#$6*
The shifted string is: THIS IS A TEST! 12#$6*
```

7-11 A subroutine to return the positions of the first and last non-blank characters in a string is shown below. Note that this subroutine is designed to return `ibeg` = `iend` = 1 whenever a string is completely blank. This choice is arbitrary.

```
SUBROUTINE string_limits ( string, ibeg, iend )
!
!  Purpose:
!    To determine the limits of the non-blank characters within
!    a character variable.  This subroutine returns pointers
!    to the first and last non-blank characters within a string.
!    If the string is completely blank, it will return ibeg =
!    iend = 1, so that any programs using these pointers will
!    not blow up.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/09/96    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of dummy arguments:
CHARACTER(len=*),INTENT(IN) :: string ! Input string
INTEGER,INTENT(OUT) :: ibeg  ! First non-blank character
INTEGER,INTENT(OUT) :: iend  ! Last non-blank character

! List of local variables:
INTEGER :: length            ! Length of input string

! Get the length of the input string.
length = LEN ( string )

! Look for first character used in string.  Use a
! WHILE loop to find the first non-blank character.
ibeg = 0
DO
   ibeg = ibeg + 1
   IF ( ibeg > length ) EXIT
```

```
   IF ( string(ibeg:ibeg) /= ' ' ) EXIT
END DO

! If ibeg > length, the whole string was blank.  Set
! ibeg = iend = 1.  Otherwise, find the last non-blank
! character.
IF ( ibeg > length ) THEN
   ibeg = 1
   iend = 1
ELSE
   ! Find last nonblank character.
   iend = length + 1
   DO
      iend = iend - 1
      IF ( string(iend:iend) /= ' ' ) EXIT
   END DO
END IF

END SUBROUTINE
```

A test driver program for subroutine `string_limits` is shown below.

```
PROGRAM test_string_limits
!
!  Purpose:
!    To test subroutine string_limits.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/09/96    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of local variables:
CHARACTER(len=30),DIMENSION(3) :: a  ! Test strings
INTEGER :: i                         ! Loop index
INTEGER :: ibeg                      ! First non-blank char
INTEGER :: iend                      ! Last non-blank char

! Initialize strings
a(1) = 'How many characters are used?'
a(2) = '   ...and how about this one?   '
a(3) = '   !   !                         '

! Write results.
DO i = 1, 3
   WRITE (*,'(/1X,A,I1,2A)') 'a(',i,')                    = ', a(i)
   CALL string_limits ( a(i), ibeg, iend )
   WRITE (*,'(1X,A,I3)')     'First non-blank character = ', ibeg
   WRITE (*,'(1X,A,I3)')     'Last non-blank character  = ', iend
END DO

END PROGRAM
```

When the program is executed, the results are:

```
C:\BOOK\F90\SOLN>test_string_limits

a(1)                      = How many characters are used?
First non-blank character =   1
Last non-blank character  =  29

a(2)                      =    ...and how about this one?
First non-blank character =   4
Last non-blank character  =  29

a(3)                      =    !   !
First non-blank character =   4
Last non-blank character  =   8
```

Chapter 8. Additional Data Types

8-1 "Kinds" are versions of the same basic data type that have differing characteristics. For the real data type, different kinds have different ranges and precisions. A Fortran compiler must support at least two kinds of real data: single precision and double precision.

8-5 *(a)* These statements are legal. They read ones into the double precision real variable `a` and twos into the single precision real variable `b`. Since the format descriptor is `F18.2`, there will 16 digits to the left of the decimal point. The result printed out by the `WRITE` statement is

```
  1.111111111111111E+015    2.222222E+15
```

(b) These statements are illegal. Complex values cannot be compared with the `>` relational operator.

8-10 A subroutine to accept a complex number C and calculate its amplitude and phase is shown below:

```
SUBROUTINE complex_2_amp_phase ( c, amp, phase )
!
!  Purpose:
!    Subroutine to accept a complex number C = RE + i IM and
!    return the amplitude "amp" and phase "phase" of the number.
!    This subroutine returns the phase in radians.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/10/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of dummy arguments:
COMPLEX,INTENT(IN) :: c           ! Input complex number
REAL,INTENT(OUT) :: amp           ! Amplitude
REAL,INTENT(OUT) :: phase         ! Phase in radians

! Get amplitude and phase.
amp   = ABS ( c )
```

```
phase = ATAN2 ( AIMAG(c), REAL(c) )

END SUBROUTINE
```

A test driver program for this subroutine is shown below:

```
PROGRAM test
!
!  Purpose:
!    To test subroutine complex_2_amp_phase, which converts an
!    input complex number into amplitude and phase components.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/10/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! Local variables:
REAL :: amp                    ! Amplitude
COMPLEX :: c                   ! Complex number
REAL :: phase                  ! Phase

! Get input value.
WRITE (*,'(A)') ' Enter a complex number:'
READ (*,*) c

! Call complex_2_amp_phase
CALL complex_2_amp_phase ( c, amp, phase )

!     Tell user.
WRITE (*,'(A,F10.4)') ' Amplitude = ', amp
WRITE (*,'(A,F10.4)') ' Phase     = ', phase

END PROGRAM
```

Some typical results from the test driver program are shown below. The results are obviously correct.

```
C:\BOOK\F90\SOLN>test
Enter a complex number:
(1,0)
Amplitude =     1.0000
Phase     =      .0000

C:\BOOK\F90\SOLN>test
Enter a complex number:
(0,1)
Amplitude =     1.0000
Phase     =     1.5708

C:\BOOK\F90\SOLN>test
Enter a complex number:
(-1,0)
Amplitude =     1.0000
Phase     =     3.1416
```

```
C:\BOOK\F90\SOLN>test
Enter a complex number:
(0,-1)
Amplitude =     1.0000
Phase     =    -1.5708
```

8-16 The definitions of "point" and "line" are shown below:

```
! Declare type "point"
TYPE :: point
   REAL :: x                    ! x position
   REAL :: y                    ! y position
END TYPE point

! Declare type "line"
TYPE :: line
   REAL :: m                    ! Slope of line
   REAL :: b                    ! Y-axis intercept of line
END TYPE line
```

8-17 A function to calculate the distance between two points is shown below. Note that it is placed in a module to create an explicit interface.

```
MODULE geometry
!
!  Purpose:
!    To define the derived data types "point" and "line".
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!     01/11/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare type "point"
TYPE :: point
   REAL :: x                    ! x position
   REAL :: y                    ! y position
END TYPE point

! Declare type "line"
TYPE :: line
   REAL :: m                    ! Slope of line
   REAL :: b                    ! Y-axis intercept of line
END TYPE line

CONTAINS

FUNCTION distance(p1,p2)
!
!  Purpose:
!    To calculate the distance between two values of type "point".
!
!  Record of revisions:
```

```
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!     01/11/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of dummy arguments:
TYPE (point),INTENT(IN) :: p1        ! First point
TYPE (point),INTENT(IN) :: p2        ! Second point
REAL :: distance                     ! Distance between points

! Calculate distance
distance = SQRT ( (p1%x - p2%x)**2 + (p1%y - p2%y )**2 )

END FUNCTION distance

END MODULE geometry
```

A test driver program for this function is:

```
PROGRAM test_distance
!
!  Purpose:
!    To test function distance.
!
!  Record of revisions:
!       Date       Programmer          Description of change
!       ====       ==========          =====================
!     01/11/97    S. J. Chapman        Original code
!
USE geometry
IMPLICIT NONE

! Declare variables:
TYPE (point) :: p1, p2                ! Points

WRITE (*,*) 'Enter first point: '
READ (*,*) p1
WRITE (*,*) 'Enter second point: '
READ (*,*) p2
WRITE (*,*) 'The result is: ', distance(p1,p2)

END PROGRAM
```

When this program is executed, the results are.

```
C:\book\f90\SOLN>test_distance
Enter first point:
0 0
Enter second point:
3 4
The result is:        5.000000

C:\book\f90\SOLN>test_distance
Enter first point:
1 -1
```

```
Enter second point:
1 1
The result is:        2.000000
```

Chapter 9. Advanced Features of Procedures and Modules

9-7 The **scope** of an object is the portion of a Fortran program over which it is defined. There are three levels of scope in a Fortran 90/95 program. They are:

1. **Global** — Global objects are objects which are defined throughout an entire program. The names of these objects must be unique within a program. Examples of global objects are the names of programs, external procedures, and modules.

2. **Local** — Local objects are objects which are defined and must be unique within a single **scoping unit**. Examples of scoping units are programs, external procedures, and modules. A local object within a scoping unit must be unique within that unit, but the object name, statement label, etc. may be reused in another scoping unit without causing a conflict. Local variables are examples of objects with local scope.

3. **Statement** — The scope of certain objects may be restricted to a single statement within a program unit. The only examples that we have seen of objects whose scope is restricted to a single statement are the implied `DO` variable in an array constructor and the index variables in a `FORALL` statement.

9-10 *(a)* This statement is legal. However, *y* and *z* should be initialized before the `CALL` statement, since they correspond to dummy arguments with `INTENT(IN)`. *(c)* This statement is illegal. Dummy argument `d` is not optional, and is missing in the `CALL` statement. *(e)* This statement is illegal. Dummy argument `b` is a non-keyword argument after a keyword argument, which is not allowed.

9-12 An interface block is a construct that creates an explicit interface for an external procedure. The interface block specifies all of the interface characteristics of an external procedure. An interface block is created by duplicating the calling argument information of a procedure within the interface. The form of an interface is

```
INTERFACE
   interface_body_1
   interface_body_2
   ...
END INTERFACE
```

Each *`interface_body`* consists of the initial `SUBROUTINE` or `FUNCTION` statement of the external procedure, the type specification statements associated with its arguments, and an `END SUBROUTINE` or `END FUNCTION` statement. These statements provide enough information for the compiler to check the consistency of the interface between the calling program and the external procedure.

Alternatively, the *`interface_body`* could be a `MODULE PROCEDURE` statement if the the procedure is defined in a module.

In interface block would be needed when we want to created an explicit interface for older procedures written in earlier versions of Fortran, or for procedures written in other languages such as C.

9-23 Assess to data items and procedures in a module can be controlled using the `PUBLIC` and `PRIVATE` attributes. The `PUBLIC` attribute specifies that an item will be visible outside a module in any program unit that uses the module, while the `PRIVATE` attribute specifies that an item will not be visible to any procedure outside of the module in which the item is defined. These attributes may also be specified in `PUBLIC` and `PRIVATE` statements. By default, all objects defined in a module have the `PUBLIC` attribute.

Access to items defined in a module can be further restricted by using the ONLY clause in the USE statement to specify the items from a module which are to be used in the program unit containing the USE statement.

Chapter 10. Advanced I/O Concepts

10-1 The ES format descriptor displays a number in scientific notation, with one significant digit to the left of the decimal place, while the EN format descriptor displays a number in engineering notation, with an exponent which is a multiple of 3 and a mantissa between 1.0 and 999.9999. These difference between these two descriptors is illustrated by the following program:

```
PROGRAM test
WRITE (*,'(1X,ES14.6,/,1X,EN14.6)') 12345.67, 12345.67
END PROGRAM
```

When this simple program is executed, the results are:

```
C:\book\f90\SOLN>test
  1.234567E+04
 12.345670E+03
```

10-10 Namelist I/O is a convenient way to write out a fixed list of variable names and values, or to read in a fixed list of variable names and values. A namelist is just a list of variable names that are always read or written as a group. A NAMELIST I/O statement looks like a formatted I/O statement, except that the FMT= clause is replaced by a NML= clause. When a namelist-directed WRITE statement is executed, the names of all of the variables in the namelist are printed out together with their values in a special order. The first item to be printed is an ampersand (&) followed by the namelist name. Next comes a series of output values in the form "NAME=value". Finally, the list is terminated by a slash (/).

When a namelist-directed READ statement is executed, the program searches the input file for the marker &*nl_name*, which indicates the beginning of the namelist. It then reads all of the values in the namelist until a slash character (/) is encountered to terminate the READ. The values are assigned to the namelist variables according to the names given in the input list. The namelist READ statement does not have to set a value for every variable in the namelist. If some namelist variables are not included in the input file list, then their values will remain unchanged after the namelist READ executes.

Namelist-directed READ statements are very useful for initializing variables in a program. Suppose that you are writing a program containing 100 input variables. The variables will be initialized to their usual values by default in the program. During any particular run of the program, anywhere from 1 to 10 of these values may need to be changed, but the others would remain at their default values. In this case, you could include all 100 values in a namelist and include a namelist-directed READ statement in the program. Whenever a user runs the program, he or she can just list the few values to be changed in the namelist input file, and all of the other input variables will remain unchanged. This approach is much better than using an ordinary READ statement, since all 100 values would need to be listed in the ordinary READ's input file, even if they were not being changed during a particular run.

10-15 The status of the file is 'UNKNOWN'. It is a formatted file opened for sequential access, and the location of the file pointer is 'ASIS', which is processor dependent. It is opened for both reading and writing, with a variable record length. List-directed character strings will be written to the file without delimiters. If the file is not found, the results of the OPEN are processor dependent; however, most processors will create a new file and open it. If an error occurs during the open process, the program will abort with a runtime error.

10-16 *(b)* The status of the file is 'REPLACE'. If the file does not exist, it will be created. If it does exist, it will be deleted and a new file will be created. It is an unformatted file opened for direct access. List-directed i/o dies not apply to

unformatted files, so the delimiter clause is meaningless for this file. It is opened for writing only. The length of each record is 80 processor-dependent units. If there is an error in the open process, the program containing this statement will continue, with `ISTAT` set to an appropriate error code.

Chapter 11. Pointers and Dynamic Data Structures

11-2 An ordinary assignment statement assigns a value to a variable. If a pointer is included on the right-hand side of an ordinary assignment statement, then the value used in the calculation is the value stored in the variable pointed to by the pointer. If a pointer is included on the left-hand side of an ordinary assignment statement, then the result of the statement is stored in the variable pointed to by the pointer. By contrast, a pointer assignment statement assigns the *address* of a value to a pointer variable.

In the statement "**`a = z`**", the value contained in variable `z` is stored in the variable *pointed to* by `a`, which is the variable `x`. In the statement "**`a => z`**", the *address* of variable `z` is stored in the pointer `a`.

11-7 The statements required to create a 1000-element integer array and then point a pointer at every tenth element within the array are shown below:

```
INTEGER,DIMENSION(1000),TARGET :: my_data = (/ (i, i=1,1000) /)
INTEGER,DIMENSION(:),POINTER :: ptr
ptr => my_data(1:1000:10)
```

11-11 This program has several serious flaws. Subroutine `running_sum` allocates a new variable on pointer `sum` each time that it is called, resulting in a memory leak. In addition, it does not initialize the variable that it creates. Since `sum` points to a different variable each time, it doesn't actually add anything up! A corrected version of this program is shown below:

```
MODULE my_sub
CONTAINS
   SUBROUTINE running_sum (sum, value)
   REAL, POINTER :: sum, value
   IF ( .NOT. ASSOCIATED(sum) ) THEN
      ALLOCATE(sum)
      sum = 0.
   END IF
   sum = sum + value
   END SUBROUTINE running_sum
END MODULE
PROGRAM sum_values
USE my_sub
IMPLICIT NONE
INTEGER :: istat
REAL, POINTER :: sum, value
ALLOCATE (sum, value, STAT=istat)
WRITE (*,*) 'Enter values to add: '
DO
   READ (*,*,IOSTAT=istat) value
   IF ( istat /= 0 ) EXIT
   CALL running_sum (sum, value)
   WRITE (*,*) ' The sum is ', sum
END DO
END PROGRAM
```

When this program is compiled and executed with the Lahey Fortran 90 compiler, the results are:

```
C:\book\f90\SOLN>lf90 ex11_11.f90
Lahey Fortran 90 Compiler Release 3.00b  S/N: F9354220
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.
Copyright (C) 1985-1996 Intel Corp.  All rights reserved.
Registered to:  Stephen J. Chapman

Options:
-nap           -nbind         -nc            -nchk          -co
-ndal          -ndbl          -ndll          -nf90          -nfix
-ng            -hed           -nin           -inln          -nlisk
-nlst          -nml           -o1            -out ex11_11.exe
-pca           -sav           -stack 20000h  -stchk         -nswm
-nsyn          -t4            -ntrace        -ntrap         -nvax
-vm            -w             -nwin          -nwo           -wrap
-nxref

Compiling file ex11_11.f90.
Compiling program unit MY_SUB at line 1.
Compiling program unit SUM_VALUES at line 12.
Encountered 0 errors, 0 warnings in file ex11_11.f90.

386|LIB: 7.0 -- Copyright (C) 1986-96 Phar Lap Software, Inc.
386|LINK: 8.0_Lahey2 -- Copyright (C) 1986-96 Phar Lap Software, Inc.

C:\book\f90\SOLN>ex11_11

Enter values to add: 4

 The sum is     4.00000    2

 The sum is     6.00000    5

 The sum is     11.0000    7

 The sum is     18.0000    4

 The sum is     22.0000    ^D
```

11-17 A function that returns a pointer to the largest value in an input array is shown below. Note that it is contained in a module to produce an explicit interface.

```
MODULE subs
CONTAINS

   FUNCTION maxval (array) RESULT (ptr_maxval)
   !
   !  Purpose:
   !    To return a pointer to the maximum value in a
   !    rank one array.
   !
   !  Record of revisions:
   !      Date       Programmer          Description of change
   !      ====       ==========          =====================
   !    01/25/97    S. J. Chapman        Original code
```

```
   !
   IMPLICIT NONE

   ! Declare calling arguments:
   REAL,DIMENSION(:),TARGET,INTENT(IN) :: array  ! Input array
   REAL,POINTER :: ptr_maxval           ! Pointer to max value

   ! Declare local variables:
   INTEGER :: i             ! Index variable
   REAL :: max              ! Maximum value in array

   max = array(1)
   ptr_maxval => array(1)
   DO i = 2, UBOUND(array,1)
      IF ( array(i) > max ) THEN
         max = array(i)
         ptr_maxval => array(i)
      END IF
   END DO

   END FUNCTION maxval
END MODULE
```

A test driver program for this function is shown below:

```
PROGRAM test_maxval
!
!  Purpose:
!    To test function maxval.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/25/97    S. J. Chapman        Original code
!
USE subs
IMPLICIT NONE

! Declare variables
REAL,DIMENSION(6),TARGET :: array = (/ 1., -34., 3., 2., 87., -50. /)
REAL,POINTER :: ptr

! Get pointer to max value in array
ptr => maxval(array)

! Tell user
WRITE (*,'(1X,A,F6.2)') 'The max value is: ', ptr

END PROGRAM
```

When this program is executed, the results are:

```
C:\book\f90\SOLN>test_maxval
The max value is:  87.00
```

Chapter 12. Introduction to Numerical Methods

12-1 Two subroutines to calculate the derivative of a function using the central difference method and the forward difference method are shown below.

```
SUBROUTINE deriv_cen ( f, x0, dx, dfdx, error )
!
!  Purpose:
!    To take the derivative of function f(x) at point x0
!    using the central difference method with step size dx.
!    This subroutine expects the function f(x) to be passed
!    as a calling argument.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/25/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! Declare dummy arguments:
REAL,EXTERNAL :: f           ! Function to differentiate
REAL,INTENT(IN) :: x0        ! Location to take derivative
REAL,INTENT(IN) :: dx        ! Desired step size
REAL,INTENT(OUT) :: dfdx     ! Derivative
INTEGER,INTENT(OUT) :: error ! Error flag: 0 = no error
                             !             1 = dx < 0.

! If dx <= 0., this is an error.
IF ( dx < 0. ) THEN
   error = 1
   RETURN

! If dx is specified, then calculate derivative using the
! central difference method and the specified dx.
ELSE IF ( dx > 0. ) THEN
   dfdx = (f(x0+dx/2.) - f(x0-dx/2.) ) / dx
   error = 0
END IF

END SUBROUTINE

SUBROUTINE deriv_fwd ( f, x0, dx, dfdx, error )
!
!  Purpose:
!    To take the derivative of function f(x) at point x0
!    using the forward difference method with step size dx.
!    This subroutine expects the function f(x) to be passed
!    as a calling argument.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/25/97    S. J. Chapman        Original code
```

```
!
IMPLICIT NONE

! Declare dummy arguments:
REAL,EXTERNAL :: f           ! Function to differentiate
REAL,INTENT(IN) :: x0        ! Location to take derivative
REAL,INTENT(IN) :: dx        ! Desired step size
REAL,INTENT(OUT) :: dfdx     ! Derivative
INTEGER,INTENT(OUT) :: error ! Error flag: 0 = no error
                             !             1 = dx < 0.

! If dx <= 0., this is an error.
IF ( dx < 0. ) THEN
   error = 1
   RETURN

! If dx is specified, then calculate derivative using the
! central difference method and the specified dx.
ELSE IF ( dx > 0. ) THEN
   dfdx = (f(x0+dx) - f(x0) ) / dx
   error = 0
END IF

END SUBROUTINE
```

A test driver program that uses these two routines to calculate the derivative of the function $f(x) = 1/x$ for the location $x_0 = 0.15$ is shown below:

```
PROGRAM test_deriv
!
!  Purpose:
!    To test subroutines deriv_cen and deriv_fwd by evaluating
!    the derivative of the function f(x) = 1. / x at
!    x0 = 0.15 for a variety of step sizes.  This program
!    compares the performance of the central difference
!    method and the forward difference method.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/25/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of external functions:
REAL,EXTERNAL :: f            ! Function to calculate deriv of

! List of variables:
REAL :: dfdx_cen              ! Derivative of function (central)
REAL :: dfdx_fwd              ! Derivative of function (forward)
REAL :: dx                    ! Step size
INTEGER :: error              ! Error falg
INTEGER :: i                  ! Loop index

! Calculate derivative for 3 step sizes, and tell user.
```

```
DO i = 1, 3
   dx = 1. / 10**I
   CALL deriv_cen ( f, 0.15, dx, dfdx_cen, error )
   CALL deriv_fwd ( f, 0.15, dx, dfdx_fwd, error )
!
   WRITE (*,1000) dx, dfdx_cen, dfdx_fwd
   1000 FORMAT (' dx = ',F5.3,'  Central Diff = ',F10.6, &
                '  Forward Diff = ',F10.6)
END DO

END PROGRAM

FUNCTION f(x)
!
!  Purpose:
!    To evaluate the function f(x) = 1. / x
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/25/97    S. J. Chapman        Original code
!
IMPLICIT NONE

! List of dummy arguments:
REAL,INTENT(IN) :: x         ! Independent variable
REAL :: f                    ! Function result

f = 1. / x

END FUNCTION
```

When this program is executed, the results are shown below. Recall from Chapter 8 that the derivative of the function is actually -44.44444... As you can see, the central difference method does a much better job of estimating the derivative than the forward difference method, regardless of step size.

```
C:\BOOK\F90\SOLN>test_deriv
dx =  .100  Central Diff = -49.999990  Forward Diff = -26.666660
dx =  .010  Central Diff = -44.493820  Forward Diff = -41.666700
dx =  .001  Central Diff = -44.444080  Forward Diff = -44.150350
```

Chapter 13. Fortran Libraries

13-1 A *library* is a collection of compiled Fortran object modules organized into a single disk file, which is indexed for easy recovery. When a Fortran program invokes procedures that are located in a library, the linker searches the library to get the object modules for the procedures, and includes them in the final program. The procedures in a library do not have to be recompiled for use, so it is quicker to compile and link programs that use some components from libraries. In addition, libraries can be distributed in object form only, protecting the source code which someone invested so much time writing. Furthermore, libraries which are in the form of modules supply an explicit interface to catch common programming errors.

Libraries may be purchased to perform many special functions, such as mathematical or statistical calculations, graphics, signal processing, etc. They allow users to add features to their programs without having to write all of the specialized code themselves.

13-19 A program to calculate the derivative of $f(x) = \sin x$ with automatic step size selection, comparing the results to the true analytical answer.

```
PROGRAM calc_derivative_1
!
!  Purpose:
!    To calculate the derivative of the function sin(x) at 101
!    equally-spaced points between 0 and 2*pi using automatic
!    step sizes, comparing the results with the analytical
!    solution.
!
!  Record of revisions:
!      Date       Programmer          Description of change
!      ====       ==========          =====================
!    01/31/97    S. J. Chapman        Original code
!
USE booklib
IMPLICIT NONE

! List of named constants:
REAL,PARAMETER :: pi = 3.141593      ! Pi

! External function:
REAL, EXTERNAL :: fun                ! Function to differentiate

! List of variables:
REAL :: dx = 2*pi / 100.             ! Step size
INTEGER :: error                     ! Error flag
INTEGER :: i                         ! Loop index
REAL :: step = 0.0                   ! Step size
REAL :: x0                           ! Point to take derivative
REAL,DIMENSION(101) :: y             ! y data points
REAL,DIMENSION(101) :: y_true        ! Analytical solution

! Calculate data points.
DO i = 1, 101
   x0 = (i-1) * dx
   step = 0.
   CALL deriv ( fun, x0, step, y(i), error )
   y_true(i) = cos(x0)
END DO

! Comapare the resulting data.
WRITE (*,1000)
1000 FORMAT (' A comparison of the true answer versus the calculation',&
             ' with auto step sizes:')
WRITE (*,1010)
1010 FORMAT (T10,' x ',T21,'True',T32,'  Auto  '&
            /T10,'===',T21,'====',T32,'========')

DO i = 1, 101
```

```
   x0 = (i-1) * dx
   WRITE (*,1020) x0, y_true(i), y(i)
   1020 FORMAT (3X,3F12.6)
END DO

END PROGRAM

FUNCTION fun(x)
IMPLICIT NONE
REAL,INTENT(IN) :: x
REAL :: fun

! Evaluate function.
fun = sin(x)

END FUNCTION
```

When this program is executed, the results are:

```
C:\BOOK\f90\SOLN>calc_derivative1
A comparison of the true answer versus the calculation with auto step sizes:
         x          True         Auto
        ===         ====       ========
       .000000    1.000000    1.000000
       .062832     .998027     .998030
       .125664     .992115     .992106
       .188496     .982287     .982269
       .251327     .968583     .968565
       .314159     .951057     .951057
      ...
      ...
      3.015929    -.992115    -.991700
      3.078761    -.998027    -.997610
      3.141593   -1.000000    -.999582
      3.204425    -.998027    -.997610
      3.267256    -.992115    -.991700
      3.330088    -.982287    -.981877
      3.392920    -.968583    -.968179
      3.455752    -.951056    -.950659
      3.518584    -.929776    -.929388
      3.581416    -.904827    -.904449
      3.644248    -.876307    -.875941
      3.707079    -.844328    -.843975
      3.769911    -.809017    -.808679
      3.832743    -.770513    -.770191
      ...
      ...
      6.031858     .968583     .968969
      6.094690     .982287     .982680
      6.157522     .992115     .992513
      6.220354     .998027     .998429
      6.283185    1.000000    1.000404
```