THE UNIVERSITY
of MANCHESTER

Fortran 90

A Conversion Course for Fortran 77 Programmers

Source Code

M A Pettipher, G S Noland, J M Brooke

Manchester Computing Centre, University of Manchester

F Lin, S Ramsden

Manchester and North HPC T&EC

MAN T&EC
The Manchester and North HPC Training and Education Centre

Edition 3.0 July 1995

Table of Contents

1 Sources, Types and Control Structures 1

1.1 kind_int.f90 3
1.2 kind_rl.f90 4
1.3 mc_int.f90 5
1.4 mc_real.f90 6
1.5 vehicle.f90 7
1.6 circle.f90 8
1.7 rectdef.f90 10
1.8 circle1.f90 11
1.9 circle2.f90 13
1.10 circle.dat 15
1.11 circle3.f90 16
1.12 circle4.f90 18
1.13 birth1.f90 19
1.14 birth2.f90 20
1.15 sq_sum.f90 21
1.16 square.dat 22
1.17 no_days.f90 23
1.18 convert.f90 25

2 Procedures and Modules 27

2.1 opt_par.f90 29
2.2 err_main.f90 31
2.3 err_sub.f90 32
2.4 err_sol.f90 33
2.5 fibon.f90 35
2.6 maxabs.f90 37
2.7 prec.f90 39
2.8 prec_mod.f90 40
2.9 generic.f90 41

2.10 gen_mod.f90 43
2.11 swap_mod.f90 45
2.12 vec_main.f90 47
2.13 vec_mod.f90 48
2.14 mon_main.f90 50
2.15 mon_mod.f90 51

3 Array Processing 54

3.1 matrix.f90 56
3.2 section.f90 57
3.3 board.f90 59
3.4 where.f90 60
3.5 vec_subs.f90 61
3.6 reshape.f90 62
3.7 construc.f90 63
3.8 recurs.f90 65
3.9 renum.f90 66
3.10 sum4.f90 68
3.11 sum2.f 70
3.12 sum5.f90 71
3.13 odd_val.f90 72
3.14 spread1.f90 74
3.15 spread2.f90 75
3.16 munro.f90 76

4 Pointer Variables 78

4.1 p_assign.f90 79
4.2 o_assign.f90 80
4.3 status.f90 81
4.4 p_array.f90 83
4.5 p_matrix.f90 85
4.6 simple.f90 86
4.7 linklist.f90 88
4.8 polyline.f90 90
4.9 poly_mod.f90 92
4.10 polyline.dat 94

5 Input and Output 95
5.1 non_adv.f90 96
5.2 inquire.f90 97
5.3 namelist.f90 98
5.4 namelist.dat 99
5.5 edit1.f90 100
5.6 edit2.f90 101
5.7 io_spec.f90 102

6 Intrinsic Functions 104
6.1 char_int.f90 105
6.2 convert.f90 106
6.3 mod_int.f90 107
6.4 model.f90 108

1 Sources, Types and Control Structures

This chapter lists the source code of the following programs and the related data files:

- **kind_int.f90** This program uses SELECTED_INT_KIND and SELECTED_REAL_KIND to find the kind values for integer variables on the system.
- **kind_rl.f90** This program uses SELECTED_INT_KIND and SELECTED_REAL_KIND to find the kind values for real variables on the system.
- **mc_int.f90** This program uses the numeric intrinsic functions to find the machine constants for integer variables with different kind values on the system.
- **mc_real.f90** This program uses the numeric intrinsic functions to find the machine constants for real variables with different kind values on the system.
- **vehicle.f90** This program defines the derived types vreg, a vehicle registration number.
- **circle.f90** This program defines a derived type point, and uses this defined type to define another derived type circle.
- **rectdef.f90** This program defines a derived type point, and uses this defined type to define another derived type rectangle.
- **circle1.f90** This program defines a derived type of circle, and shows how to pass a user-defined type to an internal subprogram.
- **circle2.f90** This program defines a derived type of a circle, and reads in the values by components from the data file circle.dat.
- **circle.dat** This file contains the data to be read by circle2.f90.
- **circle3.f90** This program defines a derived type of a circle, and reads in the values by components from standard input.
- **circle4.f90** This program defines the derived types of point and circle, and shows how to assign a value to a derived type directly.
- **birth1.f90** This program defines a derived type date, which is used to store a date of birth.

- **birth2.f90** This program defines a derived type namedate, which is used to store a name and a date of birth.
- **sq_sum.f90** This program contains a DO construct which reads numbers from the data file square.dat, skips negative numbers (use CYCLE), adds the square root of positive numbers, and concludes if the present number is zero (use EXIT).
- **square.dat** This file contains the data to be read by sq_sum.f90.
- **no_days.f90** This program reads in a month number (between 1 and 12) and a year number and uses the CASE construct to assign the number of days for that month, taking leap years into account.
- **convert.f90** This program shows how to use DO and CASE control constructs. It reads in a character string, changes the case of letters and writes out new string.

1.1 kind_int.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    kind_int
!
!  Purpose:
!
!    This program uses SELECTED_INT_KIND to find the kind values for
!    integer variables on the system.
!
!-------------------------------------------------------------------------------

PROGRAM kind_int

  IMPLICIT NONE
  INTEGER :: i

! SELECTED_INT_KIND(R)    returns kind value for integer in range
!                         -10**R < integer < 10**R
!
! SELECTED_REAL_KIND(P,R) returns kind value for real with P decimal
!                         places of precision and exponent range R.

  WRITE(*,*)
  WRITE(*,*) '     I      Kind Value'
  WRITE(*,*) '            for int.  '
  WRITE(*,*) '            with range'
  WRITE(*,*) '            +/- 10**I '
  WRITE(*,*)

  DO i = 1, 10
    WRITE(*, '(I6, I12)') i, &
             SELECTED_INT_KIND(i)
  END DO

END PROGRAM kind_int
```


1.2 kind_rl.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    kind_real
!
!  Purpose:
!
!    This program uses SELECTED_REAL_KIND to find the kind values for
!    real variables on the system.
!
!-------------------------------------------------------------------------------

PROGRAM kind_real

  IMPLICIT NONE
  INTEGER :: i

! SELECTED_INT_KIND(R)    returns kind value for integer in range
!                         -10**R < integer < 10**R
!
! SELECTED_REAL_KIND(P,R) returns kind value for real with P decimal
!                         places of precision and exponent range R.

  WRITE(*,*)
  WRITE(*,*) '     I      Kind Value      Kind Value'
  WRITE(*,*) '            for real        for real  '
  WRITE(*,*) '              with            with    '
  WRITE(*,*) '             Prec > I       Exp > 10*I'
  WRITE(*,*)

  DO i = 1, 35
    WRITE(*, '(I6, 2(I12,4X))') i, &
             SELECTED_REAL_KIND(i,1), &
             SELECTED_REAL_KIND(1,10*i)
  END DO

END PROGRAM kind_real
```

1.3 mc_int.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    machine_constants_int
!
!  Purpose:
!
!    This program uses numeric intrinsic functions to find some machine
!    constants for integer variables with different kind values on the system.
!
!-------------------------------------------------------------------------------

PROGRAM machine_constants_int

  IMPLICIT NONE

  INTEGER, PARAMETER, DIMENSION(3) :: ikvl = (/ 1,2,3 /)      ! Cray CS6400
  INTEGER (KIND = ikvl(1)) i1
  INTEGER (KIND = ikvl(2)) i2
  INTEGER (KIND = ikvl(3)) i3

  WRITE (*,*) &
    'Machine constants for integer variables with different kind values'
  WRITE (*,*)
  WRITE (*,*) 'Default integer kind value: ', KIND(1)

  WRITE (*,'(/''   ikv   Digits           Huge    Range''/)')
  WRITE(*,'(i5,i7,i20,i5)') ikvl(1), DIGITS(i1), HUGE(i1), RANGE(i1)
  WRITE(*,'(i5,i7,i20,i5)') ikvl(2), DIGITS(i2), HUGE(i2), RANGE(i2)
  WRITE(*,'(i5,i7,i20,i5)') ikvl(3), DIGITS(i3), HUGE(i3), RANGE(i3)

END PROGRAM machine_constants_int
```


1.4 mc_real.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    machine_constants_real
!
!  Purpose:
!
!    This program uses numeric intrinsic functions to find some machine
!    constants for real variables with different kind values on the system.
!
!-------------------------------------------------------------------------------

PROGRAM machine_constants_real

  IMPLICIT NONE

  INTEGER, PARAMETER, DIMENSION(2) :: ikvl = (/ 1,2 /)          ! CRAY CS6400
  REAL (KIND = ikvl(1)) r1
  REAL (KIND = ikvl(2)) r2

  WRITE (*,*) &
    'Machine constants for real variables with different kind values'
  WRITE (*,*)
  WRITE (*,*) 'Default precision has kind value: ', KIND(1.0)

  WRITE (*,'(/''   ikv   Digits         Huge           Tiny      Range''/)')
  WRITE(*,'(I5,I7, 7X, E10.2, 6X, E10.2, I7)') ikvl(1), DIGITS(r1), HUGE(r1), &
    TINY(r1), RANGE(r1)
  WRITE(*,'(I5,I7, 7X, E10.2, 6X, E10.2, I7)') ikvl(2), DIGITS(r2), HUGE(r2), &
    TINY(r2), RANGE(r2)

END PROGRAM machine_constants_real
```

1.5 vehicle.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    vehicle
!
!  Purpose:
!
!    This program defines a derived type vreg, a vehicle registration
!    number.
!
!-------------------------------------------------------------------------------

PROGRAM vehicle

  IMPLICIT NONE
  TYPE vreg                     ! definition of vreg
    CHARACTER (LEN = 1) year
    INTEGER             number
    CHARACTER (LEN = 3) place
  END TYPE vreg

  TYPE(vreg) mycar              ! mycar typed as vreg

  mycar = vreg('D', 608, 'PVR')  ! mycar assigned

  WRITE (*, *) 'printing by components'
  WRITE (*, '(1X, A1, I3, A3)') mycar%year, mycar%number, mycar%place
  WRITE (*, *) 'printing by structure'
  WRITE (*, *) mycar

END PROGRAM vehicle
```


1.6 circle.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    circledef
!
!  Purpose
!
!    This program defines a derived type point, and uses this defined type to
!    define another derived type circle.
!
!-------------------------------------------------------------------------------

PROGRAM circledef    ! to define a derived type for a circle

  IMPLICIT NONE
  TYPE point         ! defines type point
    REAL x,y
  END TYPE point

  TYPE circle        ! defines type circle
    TYPE(point) centre
    REAL        radius
  END TYPE circle

  TYPE(point) o      ! o is type point
  TYPE(circle) one   ! one is type circle


  o%x = 1.0          ! assign o by components
  o%y = 2.0

  one%centre = o     ! centre of the circle assigned to point o

  one%radius = 0.5   ! assigns the radius of the circle

  PRINT '(/)'
  PRINT *,'Print centre of circle by components'
  WRITE (*,'(2F6.2,/)') o%x,o%y

  PRINT *,'Print centre of circle by derived type'
  WRITE (*,*) o
  PRINT '(//)'

  WRITE (*,*) 'Print circle by components'
  WRITE (*,'(3F6.2,/)') one

  PRINT *,'Print circle by derived type'
```

```
  WRITE (*,*) one
  PRINT '(//)'

END PROGRAM circledef
```


1.7 rectdef.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    rectdef
!
!  Purpose:
!
!    This program defines a derived type point, and uses this defined type to
!    define another derived type rectangle.
!
!------------------------------------------------------------------------------

PROGRAM rectdef

  IMPLICIT NONE
  TYPE point                    ! defines type point
    REAL x,y
  END TYPE point

  TYPE rectangle                ! defines type rectangle
    TYPE(point) ver1, ver2, ver3, ver4
  END TYPE rectangle

  TYPE(point) a, b, c, d        ! a,b,c,d type point
  TYPE(rectangle) poly          ! poly is type rectangle

  a = point(0.0, 0.0)
  b = point(0.0, 5.0)
  c = point(3.0, 5.0)
  d = point(3.0, 0.0)

  poly = rectangle(a, b, c, d)

  WRITE(*, *) 'Print corners of rectangle by components'
  WRITE(*, '(8F6.2,/)') a%x,a%y, b%x,b%y, c%x,c%y, d%x,d%y
  WRITE(*, *) 'Print corners of rectangle by derived type'
  WRITE(*, *) poly

END PROGRAM rectdef
```

1.8 circle1.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    circledef1
!
!  Purpose:
!
!    This program defines a derived type for a circle and shows how
!    to pass a user-defined type to an internal subprogram.
!
!-------------------------------------------------------------------------------

PROGRAM circledef1

  IMPLICIT NONE
  TYPE point         ! defines type point
    REAL x,y
  END TYPE point

  TYPE circle        ! defines type circle
    TYPE(point) centre
    REAL        radius
  END TYPE circle

  TYPE(point) o             ! o is type point
  TYPE(circle) one,double   ! one and double are type circle

  o%x = 1.0          ! initialises by components to avoid confusion
  o%y = 2.0          ! with complex constant
  one%centre = o     ! initialises centre component of circle one
  one%radius = 0.5   ! initialises the radius of circle one

  PRINT '(/)'
  PRINT *,'Print centre of circle by components'
  WRITE (*,'(2F6.2,/)') o%x,o%y

  PRINT *,'Print centre of circle by derived type'
  WRITE (*,*) o

  PRINT '(//)'
  WRITE (*,*) 'Print circle by components'
  WRITE (*,'(3F6.2,/)') one

  PRINT *,'Print circle by derived type'
  WRITE (*,*) one
  PRINT '(//)'
```



```
  CALL circlex2(one,double)

  PRINT *,'Now follows the circle after being doubled in radius'
  PRINT '(//)'

  WRITE (*,*) 'Print circle by components'
  WRITE (*,'(3F6.2,/)') double

  PRINT *,'Print circle by derived type'
  WRITE (*,*) double
  PRINT '(/)'

CONTAINS

  SUBROUTINE circlex2(one,two)

    TYPE(CIRCLE) one,two
    two%centre = one%centre
    two%radius = one%radius * 2.0

  END SUBROUTINE circlex2

END PROGRAM circledef1
```

1.9 circle2.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    circledef2
!
!  Purpose:
!
!    This program derives a defined type for a circle and
!    reads in the values by components from a data file
!    opened on channel 1.
!
!------------------------------------------------------------------------------

PROGRAM circledef2

  IMPLICIT NONE      ! no implicit typing

  TYPE point         ! defines type point
    REAL x,y
  END TYPE point

  TYPE circle        ! defines type circle
    TYPE(point) centre
    REAL        radius
  END TYPE circle

  TYPE(point) o      ! o is type point
  TYPE(circle) one   ! one is type circle

  OPEN (1,FILE='circle.dat')     ! opens file on channel 1

  READ (1,'(2F6.2)') o%x,o%y     ! reads in point o by component
  one%centre = o

  READ (1,'(F6.2)') one%radius   ! reads in radius by component

  PRINT '(/)'
  PRINT *,'Print centre of circle by components'
  WRITE (*,'(2F6.2,/)') o%x,o%y

  PRINT *,'Print centre of circle by derived type'
  WRITE (*,*) o

  PRINT '(//)'
  WRITE (*,*) 'Print circle by components'
  WRITE (*,'(3F6.2,/)') one
```



```
  PRINT *,'Print circle by derived type'
  WRITE (*,*) one
  PRINT '(//)'

END PROGRAM circledef2
```

1.10 circle.dat

```
4.80  9.70
3.86
```


1.11 circle3.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    circledef3
!
!  Purpose:
!
!    This program defines a derived type for a circle and reads in
!    the values by component from standard input
!
!-------------------------------------------------------------------------------

PROGRAM circledef3

  IMPLICIT NONE      ! no implicit typing

  TYPE point         ! defines type point
    REAL x,y
  END TYPE point

  TYPE circle        ! defines type circle
    TYPE(point) centre
    REAL        radius
  END TYPE circle

  TYPE(point) o      ! o is type point
  TYPE(circle) one   ! one is type circle

                     ! requests user to input numbers
  PRINT *,'Input a point which defines the centre of the circle'

  READ (*,*) o       ! reads in point o by component
  one%centre = o

  PRINT *,'Input the radius' ! requests user to input radius
  READ (*,*) one%radius      ! reads in radius by component

  PRINT '(/)'
  PRINT *,'Print centre of circle by components'
  WRITE (*,'(2F6.2,/)') o%x,o%y

  PRINT *,'Print centre of circle by derived type'
  WRITE (*,*) o

  PRINT '(//)'
  PRINT *,'Print circle by components'
  WRITE (*,'(3F6.2,/)') one
```

```
  PRINT *,'Print circle by derived type'
  WRITE (*,*) one
  PRINT '(//)'

END PROGRAM circledef3
```


1.12 circle4.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    circledef4
!
!  Purpose:
!
!    This program defines the derived types of point and circle,
!    and shows how to assign a value to a derived type directly.
!
!------------------------------------------------------------------------------

PROGRAM circledef4

  IMPLICIT NONE
  TYPE point         ! defines type point
    REAL x,y
  END TYPE point

  TYPE circle        ! defines type circle
    TYPE(point) centre
    REAL        radius
  END TYPE circle

  TYPE(circle) one   ! one is type circle

  ! Next statement assigns value to one (of derived type circle)

  one = circle(point(10.,20.),5.)

  PRINT '(//)'
  PRINT *,'Print circle by derived type'
  WRITE (*,*) one
  PRINT '(//)'

END PROGRAM circledef4
```

1.13 birth1.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    birthdate
!
!  Purpose:
!
!   This program defines a derived type date, which is used
!   to store a date of birth.
!
!-------------------------------------------------------------------------------

PROGRAM birthdate

  IMPLICIT NONE
  TYPE date                      ! defines type date
    INTEGER day
    CHARACTER(LEN = 3) month
    INTEGER year
  END TYPE date

  TYPE(date) mydate              ! mydate is of type date

  mydate = date(18, 'Nov', 1968) ! mydate assigned

  WRITE (*, *) 'Printing by components'
  WRITE (*, '(1X, I2, A3, I4)') mydate%day, mydate%month, mydate%year
  WRITE (*, *) 'Printing by structure'
  WRITE (*, *) mydate

END PROGRAM birthdate
```


1.14 birth2.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    name_birthdate
!
!  Purpose:
!
!   This program defines a derived type namedate, which is used
!   to store a name and a date of birth.
!
!-------------------------------------------------------------------------------

PROGRAM name_birthdate

  IMPLICIT NONE
  TYPE namedate       ! defines type date
    CHARACTER (LEN = 15) name
    INTEGER day
    CHARACTER (LEN = 3) month
    INTEGER year
  END TYPE namedate

  TYPE (namedate) :: person

  person = namedate('Neil Smith', 18, 'Nov', 1968)

  WRITE (*, *) 'Printing by components'
  WRITE (*, '(1X, A15, I2, A3, I4/)') &
         person%name, person%day, person%month, person%year
  WRITE (*, *) 'Printing by structure'
  WRITE (*, *) person

END PROGRAM name_birthdate
```

1.15 sq_sum.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    square_sum
!
!  Purpose:
!
!    This program contains a DO construct which reads numbers from
!    the data file square.dat, skips negative numbers (use CYCLE),
!    adds the square root of positive numbers, and concludes if
!    the present number is zero (use EXIT).
!
!-------------------------------------------------------------------------------

PROGRAM square_sum

  IMPLICIT NONE
  REAL sq_sum, number

  OPEN (1, FILE = 'square.dat')     ! open data file containing numbers

  sq_sum = 0.0                      ! preset sq_sum as 0

  DO
    READ (1, '(F4.2)') number       ! read number from the data file
    WRITE (*, *) 'number = ', number
    IF (number == 0.0) EXIT         ! exit if present number is 0
    IF (number <  0.0) CYCLE        ! skip the negative number
    sq_sum = sq_sum + SQRT(number)  ! add the square root
    WRITE (*,*) 'square_sum =', sq_sum
  END DO

  PRINT '(/)'
  PRINT *,'The solution is:'
  WRITE (*,*) sq_sum

END PROGRAM square_sum
```


1.16 square.dat

```
 4.00
-4.00
 9.00
16.00
-3.50
 1.00
25.00
-5.00
 0.00
 3.00
 5.00
```

1.17 no_days.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    no_days
!
!  Purpose:
!
!    This program reads in a month number (between 1 and 12) and a
!    year number and uses the CASE construct to assign the number
!    of days for that month, taking leap years into account.
!
!------------------------------------------------------------------------------

PROGRAM no_days

  IMPLICIT NONE

  INTEGER :: month, no_of_days, year
  LOGICAL :: leap_year, valid_month

  WRITE(*,*) 'Please enter month number and year'

  ! Read month number and year from terminal:
  READ (*,*) month, year

  ! Test for leap year:
  leap_year = MOD(year, 4) == 0

  ! Assign number of days via CASE construct:
  SELECT CASE (month)

    CASE (9, 4, 6, 11)          ! 30 days hath September, April, June and November
      no_of_days = 30

    CASE (1, 3, 5, 7:8, 10, 12) ! All the rest have 31 except February
      no_of_days = 31

    CASE (2)                    ! February depends on leap year:
      For_February: SELECT CASE (leap_year)

        CASE (.FALSE.)
          no_of_days = 28
        CASE (.TRUE.)
          no_of_days = 29

      END SELECT For_February
```



```
    CASE DEFAULT                ! Any other number not valid
      valid_month = .FALSE.
      WRITE(*,*) month, ' is not the number of a month'

  END SELECT

  IF (valid_month) WRITE (*,*) 'Number of days in month number ', &
                               month, ', year ', year, ' is ', no_of_days

END PROGRAM no_days
```

1.18 convert.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    convert_case
!
!  Purpose:
!
!    This program shows how to use DO and CASE control constructs.
!    It reads in a character string, changes the case of letters and
!    writes out new string.
!
!  Note:
!
!    Need to know the difference in the collation sequence of the upper
!    and lower case characters - use position of upper and lower case A:
!
!                         IACHAR('A') - IACHAR('a')
!
!-------------------------------------------------------------------------------

PROGRAM convert_case

  IMPLICIT NONE

  CHARACTER (LEN = 80) :: string
  INTEGER :: i, upper_to_lower, len_string

  upper_to_lower = IACHAR("a") - IACHAR("A")

  WRITE (*,*) 'Please enter string of up to 80 characters'

  ! Read in string (up to 80 characters):
  READ (*,'(A)') string

  ! Find length of string excluding trailing blanks
  len_string = LEN_TRIM(string)

  DO i = 1, len_string

    SELECT CASE (string(i:i))

      CASE ('A':'Z')            ! Upper case character found:
        string(i:i) = ACHAR(IACHAR(string(i:i)) + upper_to_lower)

      CASE ('a':'z')            ! Lower case character found:
        string(i:i) = ACHAR(IACHAR(string(i:i)) - upper_to_lower)
                                !  No change for any other characters
```



```
    END SELECT

  END DO

  ! Write out converted string
  WRITE (*,'(A)') string

END program convert_case
```

2 Procedures and Modules

This chapter lists the source code of the following programs:

- **opt_par.f90** This program calls a funtion to sum all of the integers between min and max. Set min and max to be optional keyword arguments which default to 1 and 10 respectively.
- **err_main.f90** This program calls an external subroutine. What is wrong with it? Rewrite it in a better way in Fortran 90.
- **err_sub.f90** This file contains an external subroutine ratio to be called by err_main.f90.
- **err_sol.f90** This program shows a better way to rewrite err_main.f90 and err_sub.f90 by using internal procedure.
- **fibon.f90** This program calls a recursive function to calculate the nth value of the Fibonacci sequence. $fib(1) = 1$, $fib(2) = 1$, $fib(i) = fib(i-1) + fib(i-2)$, i.e., $1, 1, 2, 3, 5, 8, 13, ...$
- **maxabs.f90** This program demonstrates how to define a generic procedure for finding the maximum absolute value using external procedures.
- **prec.f90** This program uses the precision module and shows how to select single or double precision by changing a single value.
- **prec_mod.f90** This module defines kind values for single and double precision real variables.
- **generic.f90** This program demonstrates how to define a generic procedures for swapping reals, integers and logicals using external procedures.
- **gen_mod.f90** This program demonstrates how to use the swap module which defines a generic procedure for swapping reals, integers, logicals and points (user derived type).
- **swap_mod.f90** This module defines a generic procedure for swapping reals, integers, logicals and points, where the type 'point' is defined with two real variables as components.
- **vec_main.f90** This program uses the vector module and shows how to apply all associated operators overloading to the variables of derived type vector.
- **vec_mod.f90** This module defines a vector type with x and y components and the associated operators '+' and '-' overloading.

- **mon_main.f90** This program uses the money module and shows how to apply all associated operators and assignment overloading to the variables of derived type money.
- **mon_mod.f90** This module demonstrates the power of modules as a way to define a derived type money and all associated operators and assignment overloading.

2.1 opt_par.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    opt_par
!
!  Purpose:
!
!    This program calls a function to sum all of the integers
!    between min and max. Set min and max to be optional keyword
!    arguments which default to 1 and 10 respectively.
!
!-------------------------------------------------------------------------------

Program opt_par

  IMPLICIT NONE
  INTEGER :: result

  ! Version 1: min and max not specified, so use default values.
  result = isum()
  WRITE (*,*) 'Result for min = 1 and max = 10 is ', result

  ! Version 2: min not specified.
  result = isum(max=5)
  WRITE (*,*) 'Result for min = 1 and max = 5  is ', result

  ! Version 3: max not specified.
  result = isum(min=5)
  WRITE (*,*) 'Result for min = 5 and max = 10 is ', result

  ! Version 4: both specified.
  result = isum(max=7,min=5)
  WRITE (*,*) 'Result for min = 5 and max = 7  is ', result

CONTAINS

  INTEGER FUNCTION isum(min, max)   ! A funtion to sum all of the integer
                                    ! between min and max.
    INTEGER, INTENT(IN), OPTIONAL :: min, max   ! Set min and max to be
                                                ! optional keyword

    INTEGER :: i, tmin, tmax

    ! If min or max not set, set to default:

    IF (PRESENT(min)) THEN
      tmin=min
```



```
    ELSE
      tmin=1
    ENDIF

    IF (PRESENT(max)) THEN
      tmax=max
    ELSE
      tmax=10
    ENDIF


    isum = 0                          ! preset isum as zero

    DO i = tmin, tmax
      isum = isum + i                 ! sum between min and max
    END DO

  END FUNCTION isum

END PROGRAM opt_par
```

2.2 err_main.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    errors
!
!  Purpose:
!
!    Look at this program.  What is wrong with it?  Rewrite it in a
!    better way in Fortran 90.
!
!-------------------------------------------------------------------------------


PROGRAM errors

! errors.f90          - finds ratio of two values

  IMPLICIT NONE

  INTEGER, PARAMETER :: iki = SELECTED_INT_KIND(4)
  INTEGER, PARAMETER :: ikr = SELECTED_REAL_KIND(10,200)

  REAL(ikr) :: ia, ib, ic, id
  INTEGER(iki) :: a, b, c, d

  a = 3_ikr
  b = 2_ikr

  ia = 3_iki
  ib = 2_iki

! Find ratios:

  CALL ratio(a, b, c, ia, ib, ic)
  CALL ratio(b, a, d, ib, ia, id)

! Multiply ratios to get unity:

  WRITE(*,*)  c*d, ic*id

END PROGRAM errors
```


2.3 err_sub.f90

```
SUBROUTINE ratio(a, b, c, ia, ib, ic)

  c = a/b
  ic = ia/ib

END
```

2.4 err_sol.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    errors_sol
!
!  Purpose:
!
!    This program shows a better way to rewrite err_main.f90 and
!    err_sub.f90 by using an internal procedure.
!
!-------------------------------------------------------------------------------


PROGRAM errors_sol

  IMPLICIT NONE

  INTEGER, PARAMETER :: iki = SELECTED_INT_KIND(4)
  INTEGER, PARAMETER :: ikr = SELECTED_REAL_KIND(10,200)

  REAL(ikr) :: ia, ib, ic, id
  INTEGER(iki) :: a, b, c, d

  a = 3_iki
  b = 2_iki

  ia = 3_ikr
  ib = 2_ikr

  ! Find ratios:

  CALL ratio(a, b, c, ia, ib, ic)
  CALL ratio(b, a, d, ib, ia, id)

  ! Multiply ratios to get unity:

  WRITE(*,*)  c*d, ic*id

CONTAINS

  SUBROUTINE ratio(a, b, c, ia, ib, ic)

    REAL(ikr) :: ia, ib, ic
    INTEGER(iki) :: a, b, c

    c = a/b
    ic = ia/ib
```



```
  END SUBROUTINE ratio

END PROGRAM errors_sol
```

2.5 fibon.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    fibonacci
!
!  Purpose:
!
!    This program calls a recursive function to calculate the nth value of
!    the Fibonacci sequence.
!    fib(1) = 1, fib(2) = 1, fib(i) = fib(i-1) + fib(i-2), i.e.,
!    1, 1, 2, 3, 5, 8, 13, ...
!
!
!------------------------------------------------------------------------------

PROGRAM fibonacci

  IMPLICIT NONE
  INTEGER :: n, nfib
  INTEGER, SAVE :: count = 0

  WRITE (*, *) 'Input n:'
  READ (*, *) n

  nfib = fib(n)
  WRITE (*, *) 'The nth value of the Fibonacci sequence, where n = ', n, &
               ', is: ', nfib
  WRITE (*, *) 'Function fib was invoked ', count, ' times.'

  CONTAINS

    RECURSIVE FUNCTION fib(n) RESULT(fib_result)

      INTEGER, INTENT(in) :: n
      INTEGER :: fib_result

      count = count + 1                    ! Increment count of function calls

      SELECT CASE (n)
        CASE (1)                           ! First value is 1
          fib_result = 1
        CASE (2)                           ! Second value is 1
          fib_result = 1
        CASE (3:)
          fib_result = fib(n-1) + fib(n-2)  ! Any others is sum of two previous
      END SELECT
```



```
      ! WRITE (*,*) 'n, fib = ', n, fib_result  ! Write out intermediate values

    END FUNCTION fib

END PROGRAM fibonacci
```

2.6 maxabs.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    generic_maxabs
!
!  Purpose:
!
!    This program demonstrates how to define a generic procedure for
!    finding the maximum absolute value using external procedures.
!
!------------------------------------------------------------------------------

PROGRAM generic_maxabs

  IMPLICIT NONE
  REAL :: a, b
  INTEGER :: ia, ib
  COMPLEX :: ca, cb

  INTERFACE maxabs              ! generic name
    REAL FUNCTION maxabsreal(u,v)
      REAL, INTENT(IN) :: u, v
    END FUNCTION maxabsreal

    INTEGER FUNCTION maxabsint(u,v)
      INTEGER, INTENT(IN) :: u, v
    END FUNCTION maxabsint

    REAL FUNCTION maxabscpx(u,v)
      COMPLEX, INTENT(IN) :: u, v
    END FUNCTION maxabscpx
  END INTERFACE

  ! maxabs two reals
  a = 1.0
  b = -2.0
  WRITE(*,*) 'Real values are          : ', a, b
  WRITE(*,*) 'Maximum absolute value is: ', maxabs(a,b)
  WRITE(*,*)

  ! maxabs two integers
  ia = 1
  ib = -2
  WRITE(*,*) 'Integer values are       : ', ia, ib
  WRITE(*,*) 'Maximum absolute value is: ', maxabs(ia,ib)
  WRITE(*,*)
```



```
  ! maxabs two complex
  ca = (1.0, 1.0)
  cb = (-2.0, -2.0)
  WRITE(*,*) 'Complex values are:       : ', ca, cb
  WRITE(*,*) 'Maximum absolute value is : ', maxabs(ca,cb)
  WRITE(*,*)

END PROGRAM generic_maxabs

REAL FUNCTION maxabsreal(u,v)   ! external procedure for real maxabs

  IMPLICIT NONE
  REAL, INTENT(IN) :: u, v

  maxabsreal = MAX(ABS(u),ABS(v))

END FUNCTION maxabsreal

INTEGER FUNCTION maxabsint(u,v) ! external procedure for integer maxabs

  IMPLICIT NONE
  INTEGER, INTENT(IN) :: u, v

  maxabsint = MAX(ABS(u),ABS(v))

END FUNCTION maxabsint

REAL FUNCTION maxabscpx(u,v)    ! external procedure complex maxabs

  IMPLICIT NONE
  COMPLEX, INTENT(IN) :: u, v

  maxabscpx = MAX(ABS(u),ABS(v))

END FUNCTION maxabscpx
```

2.7 prec.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    choose_prec
!
!  Purpose:
!
!    This program uses the precision module and shows how to select
!    single or double precision by changing a single value.
!
!-------------------------------------------------------------------------------

PROGRAM choose_prec

  ! Select desired precision from module.
  USE precision, ONLY : wp => dp        ! dp for double, sp for single

  IMPLICIT NONE
  REAL (KIND = wp) :: a

  WRITE(*, *) HUGE(a)

END PROGRAM choose_prec
```


2.8 prec_mod.f90

```
!-------------------------------------------------------------------------------
!
!  Module name:
!
!    precision
!
!  Purpose:
!
!    This module defines kind values for single and double
!    precision real variables.
!
!-------------------------------------------------------------------------------

MODULE precision

  ! Set up values for single or double precision

  INTEGER, PARAMETER :: sp = KIND(1.0E0)   ! Returns kind value for single prec
  INTEGER, PARAMETER :: dp = KIND(1.0D0)   ! Returns kind value for double prec

END MODULE precision
```

2.9 generic.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    generic
!
!  Purpose:
!
!    This program demonstrates how to define a generic procedure for
!    swapping reals, integers and logicals using external procedures.
!
!-------------------------------------------------------------------------------

PROGRAM generic

  IMPLICIT NONE
  REAL :: a, b
  INTEGER :: ia, ib
  LOGICAL :: la, lb

  INTERFACE swap ! generic name
    SUBROUTINE swapreal(u,v)
      REAL, INTENT(INOUT) :: u, v
    END SUBROUTINE swapreal

    SUBROUTINE swapint(u,v)
      INTEGER, INTENT(INOUT) :: u, v
    END SUBROUTINE swapint

    SUBROUTINE swaplog(u,v)
      LOGICAL, INTENT(INOUT) :: u, v
    END SUBROUTINE swaplog
  END INTERFACE

  ! swap two reals
  a = 1.0
  b = 2.0
  WRITE(*,*) a, b
  CALL swap(a,b)
  WRITE(*,*) a, b
  WRITE(*,*)

  ! swap two integers
  ia = 1
  ib = 2
  WRITE(*,*) ia, ib
  CALL swap(ia,ib)
  WRITE(*,*) ia, ib
```



```
  WRITE(*,*)

  ! swap two logicals
  la = .TRUE.
  lb = .FALSE.
  WRITE(*,*) la, lb
  CALL swap(la,lb)
  WRITE(*,*) la, lb
  WRITE(*,*)

END PROGRAM generic

SUBROUTINE swapreal(u,v) ! external procedure to swap two reals

  IMPLICIT NONE
  REAL, INTENT(INOUT) :: u, v
  REAL :: temp

  temp = u
  u = v
  v = temp

END SUBROUTINE swapreal

SUBROUTINE swapint(u,v) ! external procedure to swap two integers

  IMPLICIT NONE
  INTEGER, INTENT(INOUT) :: u, v
  INTEGER :: temp

  temp = u
  u = v
  v = temp

END SUBROUTINE swapint

SUBROUTINE swaplog(u,v) ! external procedure to swap two logicals

  IMPLICIT NONE
  LOGICAL, INTENT(INOUT) :: u, v
  LOGICAL :: temp

  temp = u
  u = v
  v = temp

END SUBROUTINE swaplog
```

2.10 gen_mod.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    generic_mod
!
!  Purpose:
!
!    This program demonstrates how to use a module which defines a generic
!    procedure for swapping reals, integers, logicals and points (user
!    derived type).
!
!-------------------------------------------------------------------------------

PROGRAM generic_mod

  USE swap_module ! defined in swap_module.f90

  IMPLICIT NONE
  REAL :: a, b
  INTEGER :: ia, ib
  LOGICAL :: la, lb
  TYPE(point) :: pa, pb

  ! swap two reals
  a = 1.0
  b = 2.0
  WRITE(*, *) a, b
  CALL swap(a, b)
  WRITE(*, *) a, b
  WRITE(*, *)

  ! swap two integers
  ia = 1
  ib = 2
  WRITE(*, *) ia, ib
  CALL swap(ia, ib)
  WRITE(*, *) ia, ib
  WRITE(*, *)

  ! swap two logicals
  la = .TRUE.
  lb = .FALSE.
  WRITE(*, *) la, lb
  CALL swap(la, lb)
  WRITE(*, *) la, lb
  WRITE(*, *)
```



```
  ! swap two points
  pa = point(1.0, 3.0)
  pb = point(2.0, 4.0)
  WRITE(*, *) pa, pb
  CALL swap(pa, pb)
  WRITE(*,*) pa, pb
  WRITE(*,*)

END PROGRAM generic_mod
```

2.11 swap_mod.f90

```
!-------------------------------------------------------------------------------
!
!  Module name:
!
!    swap_module
!
!  Purpose:
!
!    This module defines a generic procedure for swapping reals, integers,
!    logicals and points, where the type 'point' is defined with two real
!    variables as components.
!
!-------------------------------------------------------------------------------

MODULE swap_module

  IMPLICIT NONE

  TYPE point                    ! define type point
    REAL :: x, y
  END TYPE point

  INTERFACE swap                ! generic name
    MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
  END INTERFACE

CONTAINS

  SUBROUTINE swapreal(u,v)      ! module procedure to swap two reals

    REAL, INTENT(INOUT) :: u, v
    REAL :: temp

    temp = u
    u = v
    v = temp

  END SUBROUTINE swapreal

  SUBROUTINE swapint(u,v)       ! module procedure to swap two integers

    INTEGER, INTENT(INOUT) :: u, v
    INTEGER :: temp

    temp = u
    u = v
    v = temp
```



```
  END SUBROUTINE swapint

  SUBROUTINE swaplog(u,v)       ! module procedure to swap two logicals

    LOGICAL, INTENT(INOUT) :: u, v
    LOGICAL :: temp

    temp = u
    u = v
    v = temp

  END SUBROUTINE swaplog

  SUBROUTINE swappoint(u,v)     ! module procedure to swap two points

    TYPE(point), INTENT(INOUT) :: u, v
    TYPE(point) :: temp

    temp = u
    u = v
    v = temp

  END SUBROUTINE swappoint

END MODULE swap_module
```

2.12 vec_main.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    vector_main
!
!  Purpose:
!
!    This program uses the vector module and shows how to apply all
!    associated operators overloading to the variables of derived type
!    vector.
!
!-------------------------------------------------------------------------------

PROGRAM vector_main

  USE vector_module

  IMPLICIT NONE
  TYPE (vector) :: v, v1, v2

  WRITE(*, *) 'Assign values to v1 and v2'

  v1 = vector(1.0, 2.0)
  WRITE(*, *) 'v1 = ', v1

  v2 = vector(4.0, 5.0)
  WRITE(*, *) 'v2 = ', v2

  WRITE(*, *)
  WRITE(*, *) 'Use new vector operators'

  v = -v2
  WRITE(*, *) 'v = -v2 = ', v

  v = v1 + v2
  WRITE(*, *) 'v = v1 + v2 = ', v

  v = v1 - v2
  WRITE(*, *) 'v = v1 - v2 = ', v

END PROGRAM vector_main
```


2.13 vec_mod.f90

```
!-------------------------------------------------------------------------------
!
!  Module name:
!
!    vector_module
!
!  Purpose:
!
!    This module defines a vector type with x and y components and the
!    associated operators '+' and '-' overloading.
!
!-------------------------------------------------------------------------------

MODULE vector_module

  ! Specifications:

  TYPE vector
    REAL :: x, y
  END TYPE vector

  ! Interfaces:

  INTERFACE OPERATOR (+)        ! overloading operator +
    MODULE PROCEDURE vector_add
  END INTERFACE

  INTERFACE OPERATOR (-)        ! overloading operator -
    MODULE PROCEDURE vector_negate, vector_sub
  END INTERFACE

CONTAINS

  ! Module procedures:

  FUNCTION vector_add(v1, v2)

    TYPE (vector) :: vector_add
    TYPE (vector), INTENT(IN) :: v1, v2

    vector_add%x = v1%x + v2%x
    vector_add%y = v1%y + v2%y

  END FUNCTION vector_add

  FUNCTION vector_negate(v)

    TYPE (vector) :: vector_negate
```

```
    TYPE (vector),INTENT(IN) :: v

    vector_negate%x = -v%x
    vector_negate%y = -v%y

  END FUNCTION vector_negate

  FUNCTION vector_sub(v1, v2)

    TYPE (vector) :: vector_sub
    TYPE (vector), INTENT(IN) :: v1, v2

    vector_sub = v1 + (-v2)

  END FUNCTION vector_sub

END MODULE vector_module
```


2.14 mon_main.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    money_main
!
!  Purpose:
!
!    This program uses the money module and shows how to apply all
!    associated operators and assignment overloading to the
!    variables of derived type money.
!
!-------------------------------------------------------------------------------

PROGRAM money_main

  USE money_module

  IMPLICIT NONE
  TYPE (money) :: price, tax, net, checkprice, zero

  price = money(99, 99)

  ! Use new assignment (excluding pence):

  tax = price%pounds * vatrate
  WRITE(*,*) 'tax1 = ', tax

  ! Use new operator .VAT.:

  tax = .VAT. price
  WRITE(*,*) 'tax2 = ', tax

  ! Use new money operators:

  net = price - tax
  WRITE(*,*) 'Net price = ', net

  checkprice = net + tax
  WRITE(*,*) 'net + tax = ', checkprice

  zero = price + (-checkprice)
  WRITE(*,*) 'Check zero: ', zero

END PROGRAM money_main
```

2.15 mon_mod.f90

```
!-------------------------------------------------------------------------------
!
!  Module name:
!
!    money_module
!
!  Purpose:
!
!    This module demonstrates the power of modules as a way to define
!    a derived type money and all associated operators and assignment
!    overloading.
!
!-------------------------------------------------------------------------------

MODULE money_module

  ! Specifications:

  TYPE money
    INTEGER :: pounds, pence
  END TYPE money

  ! Interfaces:

  INTERFACE OPERATOR (+)        ! overloading operator +
    MODULE PROCEDURE addmoney
  END INTERFACE

  INTERFACE OPERATOR (-)        ! overloading operator -
    MODULE PROCEDURE negatemoney, subtractmoney
  END INTERFACE

  INTERFACE OPERATOR (.VAT.)    ! defining new operator .VAT.
    MODULE PROCEDURE calcvat
  END INTERFACE

  INTERFACE ASSIGNMENT (=)      ! overloading assignment by excluding pence
    MODULE PROCEDURE realtomoney
  END INTERFACE

  REAL :: vatrate = 0.175       ! global data

CONTAINS

  ! Module procedures:

  FUNCTION addmoney(a, b)
```



```
    TYPE (money) :: addmoney
    TYPE (money),INTENT(IN) :: a, b
    INTEGER :: carry, temppence

    temppence = a%pence + b%pence
    carry = 0

    IF (temppence>100) THEN
      temppence = temppence - 100
      carry = 1
    END IF
    addmoney%pounds = a%pounds + b%pounds + carry
    addmoney%pence = temppence

  END FUNCTION addmoney

  FUNCTION negatemoney(a)

    TYPE (money) :: negatemoney
    TYPE (money),INTENT(IN) :: a

    negatemoney%pounds = -a%pounds
    negatemoney%pence = -a%pence

  END FUNCTION negatemoney

  FUNCTION subtractmoney(a,b)

    TYPE (money) :: subtractmoney
    TYPE (money),INTENT(IN) :: a,b
    INTEGER :: temppound, temppence

    temppence = a%pence - b%pence
    temppound = a%pounds - b%pounds

      IF ((temppence<0).AND.(temppound>0)) THEN
        temppence = 100 + temppence
        temppound = temppound - 1
      ELSE IF ((temppence>0).AND.(temppound<0)) THEN
        temppence = temppence - 100
        temppound = temppound + 1
      END IF

    subtractmoney%pence = temppence
    subtractmoney%pounds = temppound

  END FUNCTION subtractmoney

  FUNCTION calcvat(x) RESULT (vat)

    TYPE (money), INTENT (IN) :: x
```

```
    TYPE (money) :: vat
    INTEGER :: taxpence

    taxpence = ceiling (vatrate * (x%pounds * 100 + x%pence))
    vat%pounds = taxpence / 100
    vat%pence = taxpence - 100 * vat%pounds

  END FUNCTION calcvat

  SUBROUTINE realtomoney(m, r)  ! excluding pence

    TYPE (money), INTENT(OUT) :: m
    REAL, INTENT(IN) :: r

    m%pounds = r                ! real to integer coercion is already available
    m%pence = floor((r - m%pounds) * 100)

  END SUBROUTINE realtomoney

END MODULE money_module
```


3 Array Processing

This chapter lists the source code of the following programs:

- **matrix.f90** This program declares a 2-dimensional integer array and uses DO loops to assign elements of the array to have values row column, e.g., $a(3,2) = 32$, $a(5,7) = 57$. Then it writes the resulting matrix to the data file matrix.dat for later uses.
- **section.f90** This program demonstrates the use of array sections to write out. The array data is read in from the file matrix.dat.
- **board.f90** This program demonstrates the use of array sections to generate an 8x8 chequerboard, with 'B' and 'W' in alternate positions.
- **where.f90** This program uses the WHERE construct to create an array containing all of the odd values and 0 elsewhere. It also uses the elemental intrinsic function MOD. The original array data is read in from the file matrix.dat.
- **vec_subs.f90** This program demonstrates the use of vector subscript. It declares a vector subscript, iv, with extent 5. From the array read in from the file matrix.dat, it creates a 5x5 array containing only the odd values.
- **reshape.f90** This program demonstrates the use of array constructor and the instrinsic function RESHAPE to generate the same array as the one created by matrix.f90.
- **construc.f90** This program demonstrates the use of array constructors.
- **recurs.f90** This program explains that the array syntax assumes no recursion and we should beware of the array recursion.
- **renum.f90** This program illustrates the fact that elements in an expression are renumbered with 1 as the lower bound in each dimension.
- **sum4.f90** This program sums all elements of a two dimensional array. It shows how to use assumed size array to remove the necessity to pass array dimension into subroutine in Fortran 77 as sum2.f.
- **sum2.f** This Fortran 77 program sums all elements of a two dimensional array by passing array dimension into subroutine.

- **sum5.f90** This program sums all elements of a two dimensional array using the array intrinsic function SUM.
- **odd_val.f90** This program creates an integer array whose size is allocated dynamically (read size from terminal), assigns odd and even values to the array (same as matrix.f90), and passes the array to a subroutine which uses an assumed shape argument and returns all odd values of the array and 0 elsewhere.
- **spread1.f90** This program shows how to use the array intrinsic function SPREAD to assign an array.
- **spread2.f90** This program shows how to use the array intrinsic function SPREAD to assign an array with values $1.0/REAL(i+j+1)$, where i is the row number and j is the column number. It also shows how to use array constructor to get the same result.
- **munro.f90** This program shows the use of intrinsic functions to generate pseudo random real number between 0 and 4404, to select all values greater than 3000 and to find the number of them, the maximum, the minimum and the average.

3.1 matrix.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    assign_matrix
!
!  Purpose:
!
!    This program declares a 2-dimensional integer array, with extents (n,n),
!    where n is set to 9 in a PARAMETER statement.
!
!    It uses DO loops to assign elements of the array to have values r c,
!    where r is the row number and c is the column number, e.g., a(3,2) = 32,
!    a(5,7) = 57. Then it writes the resulting matrix to the data file
!    matrix.dat for later use.
!
!-------------------------------------------------------------------------------

PROGRAM assign_matrix

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 9
  INTEGER, DIMENSION (n, n) :: ia
  INTEGER :: i, j

  ! Assign matrix ia with DO loops:

  WRITE(*, '("Assign with DO loops:"/)')

  DO i = 1, n
    DO j = 1, n
      ia(j,i) = i + j * 10
    END DO
  END DO

  WRITE(*,'(9I5)') (ia(i, :), i = 1, n)

  OPEN (UNIT = 1, FILE = 'matrix.dat')
  WRITE (1, *) ia
  CLOSE (UNIT = 1)

END PROGRAM assign_matrix
```

3.2 section.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    array_section
!
!  Purpose:
!
!    This program demonstrates the use of array sections.
!
!    It firstly reads in the matrix from the file matrix.dat, which is
!    generated by matrix.f90. Then it writes out:
!
!  (a) the first row
!   (b) the fifth column
!       (c) every second element of each row and column, columnwise
!           ( a(1,1), a(3,1),...)
!       (d) every second element of each row and column, rowwise
!           ( a1,1), a(1,3),...)
!       (e) the 3 non-overlapping 3x3 sub-matrices in columns 4 to 6
!
!-------------------------------------------------------------------------------

PROGRAM array_section

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 9
  INTEGER, DIMENSION (n,n) :: ia
  INTEGER :: i, j

  ! read matrix ia from the file matrix.dat generated by matrix.f90
  OPEN (UNIT = 1, FILE = 'matrix.dat')
  READ (1, *) ia
  CLOSE (1)

  WRITE(*,'(//"First row:"/)')
  WRITE(*,*) ia(1,:)

  WRITE(*,'(//"Fifth column:"/)')
  WRITE(*,*) ia(:,5)

  WRITE(*,'(//"Every second element in each row and column, columnwise:"/)')
  WRITE(*,'(5I5)') ia(::2,::2)

  WRITE(*,'(//"Every second element in each row and column, rowwise:"/)')
  WRITE(*,'(5I5)') (ia(i,::2), i = 1, n, 2)

  WRITE(*,'(//"3 non-overlapping 3x3 arrays in columns 4 to 6:"/)')
```



```
  DO i = 1, 9, 3
    WRITE(*,'(3I5)') (ia(j, 4:6), j = i, i+2)
    WRITE(*, *)
  END DO

END PROGRAM array_section
```

3.3 board.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    chequerboard
!
!  Purpose:
!
!    This program demonstrates the use of array sections.
!
!    It generates an 8x8 chequer board, with 'B' and 'W' in
!    alternate positions. Assume the first position is 'B'.
!
!-------------------------------------------------------------------------------

PROGRAM chequerboard

  IMPLICIT NONE
  CHARACTER (LEN = 1) , DIMENSION (8,8) :: chboard

  ! Version 1: 4 statements:
  WRITE(*,'(''Version 1 - 4 statements: ''//)')

  chboard (1:8:2,::2) = 'B'
  chboard (2:8:2,::2) = 'W'
  chboard (1:8:2,2::2) = 'W'
  chboard (2:8:2,2::2) = 'B'

  WRITE(*,'(8A4)') chboard

  ! Version 2: 3 statements:
  WRITE(*,'(//''Version 2 - 3 statements: ''//)')

  chboard = 'B'
  chboard (2:8:2,::2) = 'W'
  chboard (1:8:2,2::2) = 'W'

  WRITE(*,'(8A4)') chboard

END PROGRAM chequerboard
```


3.4 where.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    where_construct
!
!  Purpose:
!
!    This program demonstrates the use of WHERE construct.
!
!    It firstly reads in the matrix from the file matrix.dat, which is
!    generated by matrix.f90. Then it uses the WHERE construct
!    and the elemental function MOD to create an array containing all
!    of the odd values and 0 elsewhere.
!
!-------------------------------------------------------------------------------

PROGRAM where_construct

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 9
  INTEGER, DIMENSION (n,n) :: ia, ib
  INTEGER :: i

  ! read matrix ia from the file matrix.dat generated by matrix.f90
  WRITE(*,'(//"Read in array from the file matrix.dat:"/)')
  OPEN (UNIT = 1, FILE = 'matrix.dat')
  READ (1, *) ia
  CLOSE (1)

  ! Write out whole array, rowwise:
  WRITE(*,'(9i5)') (ia(i,:), i = 1, n)

  ! Array of odd values and zeroes using WHERE:
  WRITE(*,'(//"Array of odd values and zeroes using WHERE:"/)')

  WHERE ( MOD(ia,2) == 1)
    ib = ia
  ELSEWHERE
    ib = 0
  END WHERE

  ! Write out new array, rowwise:
  WRITE(*,'(9i5)') (ib(i,:), i = 1, n)

END PROGRAM where_construct
```

3.5 vec_subs.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    vector_subscript
!
!  Purpose:
!
!    This program demonstrates the use of vector subscript.
!
!    It declares a vector subscript, iv, with extent 5. From the array
!    constructed in matrix.f90 (read in from the file matrix.dat) it
!    creates a 9x5 array containing only the odd values.
!
!-------------------------------------------------------------------------------

PROGRAM vector_subscript

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 9
  INTEGER, DIMENSION (n,n) :: ia
  INTEGER, DIMENSION (5) :: iv
  INTEGER, DIMENSION (n,5) :: i5x5
  INTEGER :: i

  ! read matrix ia from the file matrix.dat generated by matrix.f90
  WRITE(*,'(//"Read in array from the file matrix.dat:"/)')
  OPEN (UNIT = 1, FILE = 'matrix.dat')
  READ (1, *) ia
  CLOSE (1)

  ! Write out whole array, rowwise:
  WRITE(*,'(9i5)') (ia(i,:), i = 1, n)

  ! Construct an array of odd values using vector subscript:
  WRITE(*,'(//"Construct an array of odd values using vector subscript:"/)')

  iv = (/ 1, 3, 5, 7, 9 /)
  i5x5 = ia(:,iv)

  ! Write out array of odd values, rowwise:
  WRITE(*,'(5i5)') (i5x5(i,:), i = 1, n)

END PROGRAM vector_subscript
```


3.6 reshape.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    reshape_constructor
!
!  Purpose:
!
!    This program demonstrates the use of the array constructor and
!    the intrinsic function RESHAPE to generate the same array
!    as the one created by matrix.f90.
!
!-------------------------------------------------------------------------------

PROGRAM reshape_constructor

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 9
  INTEGER, DIMENSION (n,n) :: ia
  INTEGER :: i, j

  ! Assign matrix ia with array constructor:
  WRITE(*,'(//"Assign with Array Constructor:"/)')

  ia = RESHAPE( (/ (((i + j*10), j=1,n), i=1,n) /), SHAPE = (/ n,n /) )

  ! Write out whole array, rowwise:
  WRITE(*,'(9i5)') (ia(i, :), i = 1, n)

END PROGRAM reshape_constructor
```

3.7 construc.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    array_constructor
!
!  Purpose:
!
!    This program demonstrates the use of array constructors.
!
!-------------------------------------------------------------------------------

PROGRAM array_constructor

  IMPLICIT NONE
  REAL, DIMENSION (6) :: ra
  REAL, DIMENSION (3,2) :: r2d
  REAL, DIMENSION (5,5) :: r2da
  INTEGER, DIMENSION (2) :: shape
  INTEGER :: i, j

  ! Simple list:

  ra = (/ 1.2,3.4,3.0,11.2,1.0,3.7 /)
  WRITE(*,'("ra = ",6f6.2/)') ra

  ! Specific elements from a 5x5 array:

  DO i=1,5
    DO j=1,5
      r2da(j,i) = j*10 + i ! r2da(r,c) = rc
    END DO
  END DO

  i = 2
  ra = (/ r2da(i,2:4), r2da(1:5:2,i+3) /)
  WRITE(*,'("ra = ",6f6.2)') ra

  ! Implied DO:

  ra = (/ (1.0/REAL(i), i=1,6) /)
  WRITE(*,'(/"ra = ",6f6.2)') ra

  ! Nested implied DO:

  ra = (/ (((j-1)*3 + i, i=1,3), j=1,2) /)
  WRITE(*,'(/"ra = ",6f6.2/)') ra
```



```
  ! Assign elements in two dimensional array via RESHAPE function:

  shape = (/ 3,2 /)
  r2d = RESHAPE ( (/ (((j-1)*3 + i, i=1,3), j=1,2) /), shape )
  WRITE(*,*) 'r2d is a 3x2 array:'
  DO i=1,3
    WRITE(*,'(3f6.2)') r2d(i,:)
  END DO

END PROGRAM array_constructor
```

3.8 recurs.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    recursion
!
!  Purpose:
!
!    This program explains that the array syntax assumes no recursion and
!    we should beware of array recursion.
!
!-------------------------------------------------------------------------------

PROGRAM recursion

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 5
  INTEGER, DIMENSION(n) :: ia
  INTEGER :: i


  ! Recursion in ia:

  ia = (/ 1, 20, 300, 4000, 50000 /)

  DO i=2,n
    ia(i) = ia(i-1) + ia(i)
  END DO
  WRITE(*,'(5I10)') ia


  ! Fortran 90 uses original values of ia => no recursion:

  ia = (/ 1, 20, 300, 4000, 50000 /)

  ia(2:n) = ia(1:n-1) + ia(2:n)
  WRITE(*,'(5I10)') ia

  ! Fortran 90 equivalent to the recursive loop:

  ia = (/ 1, 20, 300, 4000, 50000 /)

  ia(2:n) =  (/ (SUM(ia(1:i)),i=2,n) /)
  WRITE(*,'(5I10)') ia

END PROGRAM recursion
```


3.9 renum.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    renum
!
!  Purpose:
!
!    This program illustrates the fact that elements in an expression are
!    renumbered with 1 as the lower bound in each dimension.
!
!-------------------------------------------------------------------------------

PROGRAM renum

  IMPLICIT NONE

  REAL, DIMENSION (-3:4) :: ra
  INTEGER, DIMENSION (1) :: locmax1, locmax2

  ! Assign elements of array ra:
  ra = (/ 1.2, 3.4, 5.4, 11.2, 1.0, 3.7, 1.0, 1.0 /)

  ! Find location of maximum value of whole array:
  ! (Note value given assumes first element has location 1.)
  locmax1 = MAXLOC(ra)
  WRITE(*,*) 'Location of maximum value of whole array is: ', locmax1(1)

  ! Find location of maximum value of array section:
  locmax2 = MAXLOC(ra(1:3))
  WRITE(*,*) 'Location of maximum value of ra(1:3) is: ', locmax2(1)

  WRITE(*,*)

  ! Find value of element with maximum value of whole array from location:
  WRITE(*,*) 'Maximum value of whole array is: ', &
             ra(LBOUND(ra) + locmax1(1) - 1)

  ! Check with MAXVAL:
   WRITE(*,*) 'Maximum value of whole array using MAXVAL is: ', MAXVAL(ra)

  WRITE(*,*)

  ! Find value of element with maximum value of array section from location:
  WRITE(*,*) 'Maximum value of ra(1:3) is: ', ra(locmax2(1))
  ! ( Value is ra(1 + locmax2(1) - 1) )

  ! Check with MAXVAL:
```

```
  WRITE(*,*) 'Maximum value of ra(1:3) using MAXVAL is: ', MAXVAL(ra(1:3))

END PROGRAM renum
```


3.10 sum4.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    sum4
!
!  Purpose:
!
!    This program sums all elements of a two dimensional array. It shows
!    how to use an assumed shape array to remove the necessity to pass array
!    dimensions into the subroutine.
!
!
!-------------------------------------------------------------------------------

PROGRAM sum4

  IMPLICIT NONE
  REAL, DIMENSION (:,:), ALLOCATABLE :: ra
  INTEGER :: i, j, n1, nsum, status

  WRITE (*,'("Input matrix size:")')
  READ (*,*) n1
  ALLOCATE (ra(n1,n1), STAT = status)

  IF (status > 0) STOP 'Fail to allocate memory for allocatable array'

  DO i=1,n1
    DO j=1,n1
      ra(j,i) = (i-1)*n1 + j
    END DO
  END DO

  CALL sumsub(ra,nsum)
  WRITE (*,'("Sum for matrix of size ",I3," = ",I10)') n1,nsum

  DEALLOCATE (ra)

CONTAINS

  SUBROUTINE sumsub(ra,nsum)

    REAL, DIMENSION (:,:) :: ra
    INTEGER :: nsum, n1, i, j

    nsum = 0
    n1 = SIZE(ra,2)
    DO  i=1,n1
```

```
      DO j=1,n1
        nsum = nsum + ra(j,i)
      END DO
    END DO

    RETURN

  END SUBROUTINE sumsub

END PROGRAM sum4
```


3.11 sum2.f

```
      PROGRAM sum2
C
C     Program to sum all elements of a two dimensional array.
C     Version 2: Fortran 77
C                Correct elements passed by defining array in
C                subroutine to have first dimension the same as
C                in the main program.



      PARAMETER (n=10)
      REAL ra(n,n)
      INTEGER i, j, n1, nsum

      DO 20 i=1,n
        DO 10 j=1,n
          ra(j,i) =1000
   10   CONTINUE
   20 CONTINUE

      WRITE (*,'("Input matrix size:")')
      READ (*,*) n1

      DO 40 i=1,n1
        DO 30 j=1,n1
          ra(j,i) = (i-1)*n1 + j
   30   CONTINUE
   40 CONTINUE

      CALL sumsub(ra,n1,nsum,n)
      WRITE (*,'("Sum for matrix of size ",I3," = ",I10)') n1,nsum

      END

      SUBROUTINE sumsub(ra,n1,nsum,n)

      REAL ra(n,n1)
      INTEGER n1, nsum, i, j

      nsum = 0
      DO 20 i=1,n1
        DO 10 j=1,n1
          nsum = nsum + ra(j,i)
   10   CONTINUE
   20 CONTINUE

      RETURN
      END
```

3.12 sum5.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    sum5
!
!  Purpose:
!
!    This program sums all elements of a two dimensional array using
!    the array intrinsic function SUM.
!
!-------------------------------------------------------------------------------

PROGRAM sum5

  IMPLICIT NONE
  REAL, DIMENSION (:,:), ALLOCATABLE :: ra
  INTEGER :: nsum, n1, i, j, status

  WRITE (*,'("Input matrix size:")')
  READ (*,*) n1
  ALLOCATE (ra(n1,n1), STAT = status)

  IF (status > 0) STOP 'Fail to allocate memory for allocatable array'

  DO i=1,n1
    DO j=1,n1
      ra(j,i) = (i-1)*n1 + j
    END DO
  END DO

  nsum = SUM(ra)
  WRITE (*,'("Sum for matrix of size ",I3," = ",I10)') n1,nsum

  DEALLOCATE (ra)

END PROGRAM sum5
```


3.13 odd_val.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    odd_values
!
!  Purpose:
!
!    This program creates an integer array whose size is allocated dynamically
!    (read size from terminal), assigns odd and even values to the array (same
!    as matrix.f90), and passes the array to a subroutine which uses an
!    assumed shape argument and returns all odd values of the array and 0
!    elsewhere.
!
!-------------------------------------------------------------------------------

PROGRAM odd_values

  IMPLICIT NONE
  INTEGER, DIMENSION (:,:), ALLOCATABLE :: ia
  INTEGER :: i, j, n, status

  ! Read size of matrix and allocate array:

  WRITE (*,'(A)',ADVANCE='NO') 'Input matrix size: '
  READ (*,'(I4)') n
  ALLOCATE (ia(n,n), STAT = status)

  IF (status > 0) STOP 'Fail to allocate memory for allocatable array'

  ! Assign matrix ia with DO loops - SHAPE parameter in RESHAPE must be
  ! a constant, so cannot use variable size as in the following:
  !
  ! ia = RESHAPE( (/ (((i + j*10), j=1,n), i=1,n) /), SHAPE = (/ n,n /) )

  WRITE(*,'(//"Assign with DO loops:"/)')
  DO i=1,n
    DO j=1,n
      ia(j,i) = i + j*10
    END DO
  END DO
  DO i=1,n
    WRITE(*,'(10i5)') ia(i,:)
  END DO

  CALL odds(ia)
  WRITE(*,'(//"Array of odd values and zeroes:"/)')
  DO i=1,n
```

```
    WRITE(*,'(10i5)') ia(i,:)
  END DO

  DEALLOCATE (ia)

CONTAINS

  SUBROUTINE odds(ia)

    IMPLICIT NONE
    INTEGER, DIMENSION (:,:) :: ia

    ! Array of odd values and zeroes using WHERE:

    WHERE ( MOD(ia,2) == 1)
      ia = ia
    ELSEWHERE
      ia = 0
    END WHERE

  END SUBROUTINE odds

END PROGRAM odd_values
```


3.14 spread1.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    spread
!
!  Purpose:
!
!    This program shows how to use the array intrinsic function SPREAD to
!    assign an array.
!
!-------------------------------------------------------------------------------

PROGRAM spread1

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 5
  INTEGER :: i
  REAL, DIMENSION (n,n) :: ra

  WRITE(*,'("Using SPREAD with DIM = 1:"//)')
  ra = REAL( SPREAD( (/ (i, i = 1, n) /), DIM = 1, NCOPIES = n) )
  WRITE(*,'(5F8.2)') (ra(:, i), i = 1, n)

  WRITE(*,'("Using SPREAD with DIM = 2:"//)')
  ra = REAL( SPREAD( (/ (i, i = 1, n) /), DIM = 2, NCOPIES = n) )
  WRITE(*,'(5F8.2)') (ra(:, i), i = 1, n)

END PROGRAM spread1
```

3.15 spread2.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    spread
!
!  Purpose:
!
!    This program shows how to use the array intrinsic function SPREAD to
!    assign an array with values 1.0/REAL(i+j+1), where i is the row number
!    and j is the column number. It also shows how to use array constructor
!    to get the same result.
!
!-------------------------------------------------------------------------------

PROGRAM spread2

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 5
  INTEGER :: i, j
  REAL, DIMENSION (n,n) :: ra

  WRITE(*, '("Using SPREAD:"//)')
  ra = 1.0 / REAL( SPREAD( (/ (i, i = 1, n) /), DIM = 1, NCOPIES = n)  &
           + SPREAD( (/ (j, j = 1, n) /), DIM = 2, NCOPIES = n) + 1 )
  WRITE(*, '(5F8.2)') (ra(:, j), j = 1, n)

  ! Using Array Constructor:

  WRITE(*,'(//"Using Array Constructor:"//)')
  ra = RESHAPE ( (/ ((1.0/(i+j+1), i=1, n), j=1, n) /), SHAPE=(/ n, n /) )
  WRITE(*,'(5F8.2)') (ra(:, j), j = 1, n)


  ! Note the simpler method with FORALL statement in Fortran 95:
  !  ??? FORALL (i = 1, n, j = 1, n) ra(i, j) = 1.0/(i+j+1)

END PROGRAM spread2
```


3.16 munro.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    munro
!
!  Purpose:
!
!    This program demonstrates the use of intrinsic functions to generate
!    pseudo random real number between 0 and 4404, and then to select
!    all values greater than 3000 and find the number of them, the maximum,
!    the minimum and the average.
!
!-------------------------------------------------------------------------------

PROGRAM munro

  IMPLICIT NONE

  REAL, DIMENSION (5000) :: hills
  LOGICAL, DIMENSION (5000) :: bighills
  INTEGER, DIMENSION (1) :: iseed
  INTEGER :: nmunros, status

  REAL, ALLOCATABLE, DIMENSION (:) :: munros

  ! Generate pseudo random numbers.

  iseed = 1
  CALL RANDOM_SEED (PUT=iseed)
  CALL RANDOM_NUMBER (hills(:))
  hills(:) = 4404.0*hills(:)

  ! Find number of Munros and allocate array munro:

  bighills = hills(:) > 3000.0                  ! Alternative to WHERE
  nmunros = COUNT(bighills)                     ! Number of Munros

  ALLOCATE (munros(nmunros), STAT = status)

  IF (status > 0) STOP 'Fail to allocate memory for allocatable array'

  ! Pack Munros into array munros:

  munros(:) = PACK(hills(:),bighills,munros(:))

  ! Print number of, highest, lowest and average heights of Munros:
```

```
  WRITE (*,'(''Number of Munros : '',I6)') nmunros
  WRITE (*,'(''Highest Munro    : '',F8.3)') MAXVAL(munros(:))
  WRITE (*,'(''Lowest Munro     : '',F8.3)') MINVAL(munros(:))
  WRITE (*,'(''Average Munro    : '',F8.3)') SUM(munros(:))/nmunros

  DEALLOCATE (munros)

END PROGRAM munro
```


4 Pointer Variables

This chapter lists the source code of the following programs and the related data files:

- **p_assign.f90** This program illustrates the use of pointer assignment statements.
- **o_assign.f90** Contrast with the program p_assign.f90, this program illustrates pointer assignments vs ordinary assignments.
- **status.f90** This program shows the status of pointers at different stages.
- **p_array.f90** This program shows the use of array pointers (the target of a pointer can also be an array and such a pointer can be referred to as an array pointer).
- **p_matrix.f90** This program shows how an array of pointers can be simulated by means of a derived type having a pointer component of the desired type, and then creating an array of that derived type.
- **simple.f90** This program demonstrates how to build and transverse a simple linked list by using pointers and dynamic allocation facilities. Notice the linked list built up stores the typed-in numbers in reverse order.
- **linklist.f90** This program demonstrates how to build and transverse a simple linked list by using pointers and dynamic allocation facilities. Notice the linked list built up preserves the order of typed-in numbers.
- **run_polyline.f90** This program uses the module polyline to build and transverse a polyline linked list containing connected vertices. It reads the polyline data from the file polyline.dat and the given y value from the terminal.
- **poly_mod.f90** This module defines point and vertex data types and contains two module procedures: delentry to delete the vertex which contains the given y coordinate and print_list to transverse the polyline to print x and y coordinates of all vertices.
- **polyline.dat** This file contains the coordinate data of points to be read by run_polyline.f90.

4.1 p_assign.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    pointer_assignment
!
!  Purpose:
!
!    This program illustrates the use of pointer assignment
!    statements.
!
!-------------------------------------------------------------------------------

PROGRAM pointer_assignments

  IMPLICIT NONE
  REAL, POINTER :: p1, p2
  REAL, TARGET :: t1 = 3.4, t2 = 4.5

  p1 => t1                  ! p1 points to t1
  PRINT *, p1, t1           ! all 3.4

  p2 => t2                  ! p2 points to t2
  PRINT *, p2, t2           ! all 4.5

  p2 => p1                  ! p2 points to the target of p1
                            ! t2 reamins unchange
  PRINT *, p1, t1, p2, t2   ! p1, t1 and p2 all 3.4, t2 remains 4.5

END PROGRAM pointer_assignments
```


4.2 o_assign.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    vs_ordinary_assignments
!
!  Purpose:
!
!    Contrast with the program p_assign.f90,
!    this program illustrates pointer assignments vs ordinary
!    assignments.
!
!-------------------------------------------------------------------------------

PROGRAM vs_ordinary_assignments

  IMPLICIT NONE
  REAL, POINTER :: p1, p2
  REAL, TARGET :: t1 = 3.4, t2 = 4.5

  p1 => t1                  ! p1 points to t1
  PRINT *, t1, p1           ! all 3.4

  p2 => t2                  ! p2 points to t2
  PRINT *, t2, p2           ! all 4.5

  p2 = p1                   ! ordinary assignment
                            ! equivalent to t2 = t1
  PRINT *, t1, p1, t2, p2   ! all 3.4

END PROGRAM vs_ordinary_assignments
```

4.3 status.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    pointer_status
!
!  Purpose:
!
!    This program shows the status of pointers at different stages
!
!  Note:
!
!    The disassociation of p did not affect q even though they were
!    both pointing at the same object. After being nullified, p can
!    be associated again either with the same or a different object later.
!    The last line just illustrates that a NULLIFY statement can
!    have more than one pointer argument.
!
!------------------------------------------------------------------------------

PROGRAM array_pointers

  IMPLICIT NONE
  REAL, POINTER :: p, q  ! undefined association status
  REAL, TARGET :: t1 = 3.4, t2 = 4.5

  p => t1 ! p points to t1
  PRINT *, "After p => t1, ASSOCIATED(p) = ", ASSOCIATED(p)     ! .T.

  q => t1 ! q also points to t1
  PRINT *, "After q => t1, ASSOCIATED(q) = ", ASSOCIATED(q)     ! .T.

  PRINT *, "ASSOCIATED(p, q) = ", ASSOCIATED(p, q)              ! .T.
  PRINT *, "but ASSOCIATED(p, t2) = ", ASSOCIATED(p, t2)        ! .F.

  NULLIFY(p)
  PRINT *, "After NULLIFY(p), ASSOCIATED(p) = ", ASSOCIATED(p)  ! .F.

  PRINT *, "ASSOCIATED(p, q) = ", ASSOCIATED(p, q)              ! .F.
  ! The disassociation of p did not affect q even though
  ! they were both pointing at the same object.
  PRINT *, "ASSOCIATED(q, t1) = ", ASSOCIATED(q, t1)            ! .T.

  ! After being nullified, p can be associated again either
  ! with the same or different object later
  p => t2 ! p points to t2
  PRINT *, "After p => t2, ASSOCIATED(p) = ", ASSOCIATED(p)     ! .T.
```



```
  ! a NULLIFY statement can have more than one pointer argument
  NULLIFY(p, q)

END PROGRAM array_pointers
```

4.4 p_array.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    array_pointers
!
!  Purpose:
!
!    This program shows the use of array pointers (the target
!    of a pointer can also be an array and such a pointer can
!    be referred to as an array pointer).
!
!-------------------------------------------------------------------------------

PROGRAM array_pointers

  IMPLICIT NONE
  REAL, DIMENSION (:), POINTER :: pv1
  REAL, DIMENSION (:, :), POINTER :: pv2
  REAL, DIMENSION (-3:5), TARGET :: tv1
  REAL, DIMENSION (5, 10), TARGET :: tv2
  INTEGER :: i

  ! Construction of tv1 array
  tv1 = (/ (i, i = 1, 9) /)
  WRITE(*, "(' tv1(-3:5) =', 9F5.1/)") tv1

  ! construction of tv2 array
  tv2 = RESHAPE((/ (i, i = 1, 50) /), SHAPE = (/ 5, 10 /))
  WRITE(*, *) "tv2(1:5, 1:10) ="
  DO I = 1, 5
    WRITE(*, "(10F5.1)") tv2(i, :)
  END DO
  !  WRITE(*, "(10F5.1)") (tv2(i, :), i = 1, 5)
  WRITE(*, *)

  pv1 => tv1 ! pv1 aliased to tv1
  WRITE(*, *) "With pv1 => tv1, pv1 has SIZE = ", SIZE(pv1), &
      ", LBOUND = ", LBOUND(pv1), ", SHAPE = ", SHAPE(pv1)
  WRITE(*, "(' pv1(-3:5) =', 9F5.1/)") pv1

  pv1 => tv1(:)         ! pv1 points to tv1 with array section subscript,
                        ! which means pv1 points to an array section, so
                        ! the lower bound is renumbered as 1
  WRITE(*, *) "With pv1 => tv1(:), pv1 has SIZE = ", SIZE(pv1), &
      ", LBOUND = ", LBOUND(pv1), ", SHAPE = ", SHAPE(pv1)
  WRITE(*, "(' pv1(1:9) =', 9F5.1/)") pv1
```



```
  pv1 => tv2(4, :)      ! pv1 points to the 4th row of tv2

  WRITE(*, *) "With pv1 => tv2(4, :), pv1 has SIZE = ", SIZE(pv1), &
      ", LBOUND = ", LBOUND(pv1), ", SHAPE = ", SHAPE(pv1)
  WRITE(*, "(' pv1(1:10) =', 10F5.1/)") pv1

  pv2 => tv2(2:4, 4:8)  ! pv2 points to a section of tv2

  WRITE(*, *) "With pv2 => tv2(2:4, 4:8), pv2 has SIZE = ", SIZE(pv2), &
      ", LBOUND = ", LBOUND(pv2), ", SHAPE = ", SHAPE(pv2)
  WRITE(*, *) "pv2(1:3, 1:5) ="
  WRITE(*, "(5F5.1)") (pv2(i, :), i = 1, 3)
  WRITE(*, *)

  pv1 => tv1(1:5:2)     ! pv1 points to a section of tv1 using triple subscript

  WRITE(*, *) "With pv1 => tv1(1:5:2), pv1 has SIZE = ", SIZE(pv1), &
      ", LBOUND = ", LBOUND(pv1), ", SHAPE = ", SHAPE(pv1)
  WRITE(*, "(' pv1(1:3) =', 3F5.1)") pv1

END PROGRAM array_pointers
```

4.5 p_matrix.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    array_of_pointers
!
!  Purpose:
!
!    This program shows how an array of pointers can be simulated
!    by means of a derived type having a pointer component of the
!    desired type, and then creating an array of that derived type.
!
!------------------------------------------------------------------------------

PROGRAM main

  ! Suppose an array of pointers to reals is required.
  ! A derived type real_pointer can be defined, whose only
  ! component is a pointer to reals
  TYPE real_pointer
    REAL, DIMENSION(:), POINTER :: p
  END TYPE real_pointer

  ! Then an array of variables of this type can be defined
  INTEGER, PARAMETER :: n=10
  TYPE(real_pointer), DIMENSION(n) :: a
  INTEGER :: i, j

  ! The following shows each row of a lower-triangular matrix
  ! may be represented by a dynamic array of increasing size.
  ! The ith pointer can be referred to by a(i)%p.
  DO i = 1, n
    ALLOCATE (a(i)%p(i))  ! allocate space for each row
    a(i)%p = (/ (j, j = 1, i) /) ! construct array for each row
    WRITE(6, "(10F5.1)") a(i)%p  ! print values of each row
  END DO

END PROGRAM
```


4.6 simple.f90

```
!------------------------------------------------------------------------------
!
!  Program name:
!
!    simple_linked_list
!
!  Purpose:
!
!    This program demonstrates how to build and transverse a
!    simple linked list by using pointers and dynamic allocation
!    facilities.
!
!    Notice the linked list stores the typed-in numbers in
!    reverse order.
!
!------------------------------------------------------------------------------

PROGRAM simple_linked_list

  IMPLICIT NONE
  TYPE node
    INTEGER :: value                    ! data filed
    TYPE (node), POINTER :: next        ! pointer field
  END TYPE node
  INTEGER :: num, status
  TYPE (node), POINTER :: list, current

  ! build up the list

  NULLIFY(list)                         ! initially nullify list (empty)

  PRINT *, 'Type-in an integer number to build a linked list (0 to terminate)'

  DO
    READ *, num                         ! read num from keyboard
    IF (num == 0) EXIT                  ! until 0 is entered
    ALLOCATE(current, STAT = status)    ! create new node
    IF (status > 0) STOP 'Fail to allocate a new node'
    current%value = num                 ! give the value
    current%next => list                ! point to previous one
    list => current                     ! update head of list
  END DO

  ! transverse the list and print the values

  PRINT *, 'Transverse the list built up and print the values'

  current => list                       ! make current as alias of list
```

```
  DO
    IF (.NOT. ASSOCIATED(current)) EXIT ! exit if null pointer
    PRINT *, current%value              ! print the value
    current => current%next             ! make current alias of next node
  END DO

END PROGRAM simple_linked_list
```


4.7 linklist.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    linked_list
!
!  Purpose:
!
!    This program demonstrates how to build and transverse a
!    simple linked list by using pointers and dynamic allocation
!    facilities.
!
!    Notice the linked list preserves the order of typed-in numbers.
!
!-------------------------------------------------------------------------------

PROGRAM linked_list

  IMPLICIT NONE
  TYPE node
    INTEGER :: value                    ! data filed
    TYPE (node), POINTER :: next        ! pointer field
  END TYPE node
  INTEGER :: num, status
  TYPE (node), POINTER :: head, tail, current

  ! build up the list

  NULLIFY(head, tail)                   ! initially nullify the list (empty)

  PRINT *, 'Type-in an integer number to build a linked list (0 to terminate)'

  READ *, num                           ! read num from keyboard
  IF (num /= 0) then                    ! if 0 is entered, do nothing
    ALLOCATE(head, STAT = status)       ! create the head of the list
    IF (status > 0) STOP 'Fail to allocate a new node'
    head%value = num                    ! give the value
    NULLIFY(head%next)                  ! point to null
    tail => head                        ! update tail of list

    DO                                  ! create rest of list
      READ *, num                       ! read num from keyboard
      IF (num == 0) EXIT                ! until 0 is entered
      ALLOCATE(current, STAT = status)  ! create new node
      IF (status > 0) STOP 'Fail to allocate a new node'
      current%value = num               ! giving the value
      NULLIFY(current%next)             ! point to null (end of list)
      tail%next => current              ! link to tail of list
```

```
      tail => current                  ! update tail of list
    END DO
  END IF

  ! transverse the list and print the values

  PRINT *, 'Transverse the list built up and print the values'

  current => head                      ! make current as alias of list
  DO
    IF (.NOT. ASSOCIATED(current)) EXIT ! exit if null pointer
    PRINT *, current%value             ! print the value
    current => current%next            ! make current alias of next node
  END DO

END PROGRAM linked_list
```


4.8 polyline.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    run_polyline
!
!  Purpose:
!
!    This program uses the module polyline to build and transverse a
!    polyline linked list containing connected vertices.
!
!-------------------------------------------------------------------------------

PROGRAM run_polyline

  USE polyline                         ! makes all module definitions available

  IMPLICIT NONE

  REAL :: yval                         ! the y coordinate of a point
  TYPE(point) :: pt
  TYPE(vertex), POINTER :: current, first
  INTEGER :: status

  ! open files to read points and y value

  OPEN (1, FILE = 'polyline.dat')

  ! build up a polyline list

  NULLIFY(first)                       ! initially nullify list (empty)

  DO
    READ(1, *) pt                      ! read point from channel 1 until 999.999
    IF (ABS(pt%x - 999.999) < 0.001) EXIT
    ALLOCATE(current, STAT = status) ! create new node
    IF (status > 0) STOP 'Fail to allocate a new vertex'
    current%pt = pt                    ! giving the point
    current%link => first              ! point to previous one
    first => current                   ! update head of polyline list
  END DO

  WRITE(*, *) 'Printing the list for the first time'
  CALL print_list(first)

  WRITE(*, *)
  WRITE(*, *) 'Input the y value of the point to be deleted from the list'
  READ (*, *) yval
```

```
  WRITE(*, *)
  WRITE(*, *) 'Looking up the list and deleting the node with the y value'
  CALL delentry(yval, first)

  WRITE(*, *)
  WRITE(*, *) 'Here is the new list:'
  CALL print_list(first)

END PROGRAM run_polyline
```


4.9 poly_mod.f90

```
!-------------------------------------------------------------------------------
!
!  Module name:
!
!    polyline
!
!  Purpose:
!
!    This module defines point and vertex data types and contains two
!    module procedures:
!
!      delentry - delete the vertex which contains the given y
!                 coordinate
!
!      print_list - transverse the polyline to print x and y coordinates
!                   of all vertices
!
!-------------------------------------------------------------------------------

MODULE polyline

  IMPLICIT NONE

  TYPE point                            ! point type definition
    REAL x,y
  END TYPE POINT

  TYPE vertex                           ! vertex type definition
    TYPE(point) pt
    TYPE(vertex), POINTER :: link
  END TYPE vertex

CONTAINS

  ! Module procedures

  ! delentry - delete the vertex which contains the given y coordinate

  SUBROUTINE delentry(yval, head)

    REAL yval
    TYPE(vertex), POINTER :: head
    TYPE(vertex), POINTER :: start, previous

    start => head
    DO
      IF (.NOT. ASSOCIATED(start)) EXIT         ! meet tail of list
      IF (ABS(start%pt%y - yval) < 0.01) EXIT   ! match the given y value
```

```
      previous => start
      start => start%link
    END DO

    IF (ASSOCIATED(start)) THEN                 ! y value found
      IF (ASSOCIATED(head, start)) THEN         ! it is in head of list
        head => start%link
      ELSE                                      ! not head of list
        previous%link => start%link
      ENDIF
      DEALLOCATE(start)
    ENDIF

  END SUBROUTINE delentry

  ! print_list - transverse the polyline to print x and y coordinates
  !              of all vertices

  SUBROUTINE print_list(head)

    TYPE(vertex), POINTER :: head
    TYPE(vertex), POINTER :: current

    current => head
    DO
      IF (.NOT. ASSOCIATED(current)) EXIT       ! tail of list
      WRITE (*,*) current%pt
      current => current%link
    END DO

  END SUBROUTINE print_list

END MODULE polyline
```


4.10 polyline.dat

```
1.1 2.2
3.3 4.4
5.5 6.6
7.7 8.8
9.9 10.10
999.999 999.999
```

5 Input and Output

This chapter lists the source code of the following programs and the related data files:

- **non_adv.f90** This program shows the use of non advancing I/O to suppress line feed after input prompt. It also shows the use of allocatable array and array intrinsic function SPREAD.
- **inquire.f90** This program shows the use of INQUIRE-by-output-list statement to determine the record length of an unformatted output list.
- **namelist.f90** This program shows the use of NAMELIST for input and output. It reads the data file namelist.dat.
- **namelist.dat** This file contains the data to be read by namelist.f90.
- **edit1.f90** This program compares the differences between E, EN, ES and G edit descriptors for output.
- **edit2.f90** This program compare the differences between I, B, O and Z edit descriptors for integer output.
- **io_spec.f90** This program demonstrates non-advancing I/O and the new I/O statement specifiers: POSITION, ACTION, ADVANCE, SIZE.

5.1 non_adv.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    non_advancing_io
!
!  Purpose:
!
!    This program shows the use of non advancing I/O to suppress
!    line feed after the input prompt.
!
!    This program also shows the use of an allocatable array and
!    array intrinsic function SPREAD.
!
!-------------------------------------------------------------------------------

PROGRAM non_advancing_io

  IMPLICIT NONE
  INTEGER :: n, i
  INTEGER, DIMENSION (:,:), ALLOCATABLE :: ia

  ! suppress line feed after input prompt by non-advancing io
  WRITE(*, '("Extent of matrix? ")', ADVANCE = 'NO')

  READ(*, '(I4)') n ! read in rank of matrix

  ALLOCATE (ia(n, n)) ! allocate array ia

  ! SPREAD intrinsic function is used to construct an array
  ! by duplicating an array section
  ia = SPREAD ( (/ (i, i = 1, n) /), DIM = 1, NCOPIES = n) &
     + SPREAD ( (/ (i, i = 1, n) /), DIM = 2, NCOPIES = n)

  ! output the values of the array ia
  DO i = 1, n
    WRITE(*, '(10I5)') ia(:, i)
  END DO

END PROGRAM non_advancing_io
```

5.2 inquire.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    inquire_iolength
!
!  Purpose:
!
!    This program shows the use of INQUIRE-by-output-list
!    statement to determine the record length of an unformatted
!    output list.
!
!-------------------------------------------------------------------------------

PROGRAM inquire_iolength

  IMPLICIT NONE

  CHARACTER (LEN = 4) :: title = 'Mr.'
  CHARACTER (LEN = 15) :: name = 'Neil Smith'
  INTEGER :: age = 32
  CHARACTER (LEN = 35) :: address = '  2 Oak Lane, Manchester M23 4QD'
  INTEGER :: tel = 4455678

  INTEGER :: rec_len ! record length to be inquired

  PRINT *, title, name, age, address, tel

  ! Assign the inquired record length to rec_len
  INQUIRE (IOLENGTH = rec_len) title, name, age, address, tel
  PRINT *, "The inquired record length = ", rec_len

  ! Open file for unformmated output using the inquired record length
  OPEN(UNIT = 1, FILE = 'temp.dat', RECL = rec_len, FORM = 'UNFORMATTED')
  WRITE(1) title, name, age, address, tel
  CLOSE(UNIT = 1)

  ! Open file for unformmated input
  OPEN(UNIT = 1, FILE = 'temp.dat', FORM = 'UNFORMATTED')
  READ(1) title, name, age, address, tel
  CLOSE(UNIT = 1)

  ! Print the list again (should be same as the first print)"
  PRINT *, title, name, age, address, tel

END PROGRAM
```


5.3 namelist.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    namelist
!
!  Purpose:
!
!    This program shows the use of NAMELIST for input and output
!
!  Note:
!
!    This program reads the data file namelist.dat.
!
!    In OPEN statements, the DELIM specifier is used, which is
!    necessary to delimit any character strings so that the
!    output sent to a file could be read subsequently by another
!    program, or even by the same one, using directed or namelist
!    formatting.
!
!-------------------------------------------------------------------------------

PROGRAM nmlist

  IMPLICIT NONE
  INTEGER :: quantity
  CHARACTER(LEN=6) :: colour(4)=(/ 'red   ', 'black ', 'green ', 'blue  ' /)
  CHARACTER(LEN=6) :: size(3)=(/ ' small', 'medium', ' large' /)
  NAMELIST /clothes/ quantity, colour, size ! define namelist group

  OPEN(6, DELIM = "APOSTROPHE")
  WRITE(6, NML=clothes) ! use namelist name as format for output

  OPEN(1, FILE='namelist.dat', DELIM = "APOSTROPHE")
  READ(1, NML=clothes) ! use namelist name as format for input
  CLOSE(1)

  WRITE(6, NML=clothes) ! output the newly read namelist
  CLOSE(6)

END PROGRAM nmlist
```

5.4 namelist.dat

```
 &CLOTHES  QUANTITY = 0, COLOUR = 'glod  ',  'yellow',  'white ',  'cyan  ', SIZE
= ' small',  'medium',  ' large' /
```


5.5 edit1.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    e_en_es_g_compare
!
!  Purpose:
!
!    To compare the differences between E, EN, ES and G
!    edit descriptors for output
!
!  Output result:
!
!    E:      0.123E+01    -0.500E+00     0.678E-02     0.988E+05
!    EN:     1.234E+00  -500.000E-03     6.780E-03    98.765E+03
!    ES:     1.234E+00    -5.000E-01     6.780E-03     9.877E+04
!    G:       1.23        -0.500         0.678E-02     0.988E+05
!
!-------------------------------------------------------------------------------

PROGRAM e_en_es_g_compare

  IMPLICIT NONE
  REAL, DIMENSION(4) :: x = (/1.234, -0.5, 0.00678, 98765.4/)

  ! print array x
  PRINT '("E: ", 4E14.3)', x    ! the 1st row is E  output
  PRINT '("EN:", 4EN14.3)', x   ! the 2nd row is EN output
  PRINT '("ES:", 4ES14.3)', x   ! the 3rd row is ES output
  PRINT '("G: ", 4G14.3)', x    ! the 4th row is G  output

END PROGRAM e_en_es_g_compare
```

5.6 edit2.f90

```
!-----------------------------------------------------------------------
!
!  Program name:
!
!    i_b_o_z_compare
!
!  Purpose:
!
!    To compare the differences between I, B, O and Z
!    edit descriptors for integer output
!
!  Output result:
!
!  I:      1         8        15       245
!  B:      1      1000      1111  11110101
!  O:      1        10        17       365
!  Z:      1         8         F        F5
!
!  I:                                                                  -15
!  B: 11111111111111111111111111111111111111111111111111111111111110001
!  O:                                          1777777777777777777761
!  Z:                                                FFFFFFFFFFFFFFF1
!
!-----------------------------------------------------------------------

PROGRAM i_b_o_z_compare

  IMPLICIT NONE
  INTEGER, DIMENSION(4) :: x = (/1, 8, 15, 245/)
  INTEGER :: i = -15

  ! print array x, 1st row is I output
  !                2nd row is B output
  !                3rd row is O output
  !                4th row is Z output

  PRINT '("I:", 4I10/"B:", 4B10/"O:", 4O10/"Z:", 4Z10)', x, x, x, x

  ! print i, note that it is negative number and
  ! EL98 represent an integer with 64 bits by default.
  ! The correspondence of rows is same as the above.

  PRINT '(/"I: ", I64/"B: ", B64/"O: ", O64/"Z: ", Z64)', i, i, i, i

END PROGRAM i_b_o_z_compare
```


5.7 io_spec.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    io_specifier
!
!  Purpose:
!
!    To demonstrate some of the new I/O facilities of Fortran 90:
!      o New statement specifiers: POSITION, ACTION, ADVANCE, SIZE
!      o Non-advancing I/O
!
!  Note:
!
!    This program will write to or read from the data file
!    io_specifier.dat
!
!-------------------------------------------------------------------------------

PROGRAM io_specifier

  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 10
  INTEGER :: i, isize
  INTEGER, DIMENSION (n) :: ix

  OPEN (7, FILE = 'io_specifier.dat', &
   POSITION = 'ASIS',         &
   ACTION = 'WRITE')

  ! Write 10 values to unit 7:
  WRITE(7, '(10I5)') (i, i = 1, n)

  CLOSE(7)

  ! Rewind the file
  OPEN (7, FILE = 'io_specifier.dat', &
   POSITION = 'REWIND',       &
   ACTION = 'READWRITE')

  ! Read first 5 elements of first record:
  READ(7, '(5I5)', ADVANCE = 'NO', SIZE = isize) (ix(i), i = 1, 5)
  PRINT *, "isize = ", isize, "  ix(1:10) = ", ix

  ! Read next 3 elements of first record:
  READ(7, '(3I5)', ADVANCE = 'NO', SIZE = isize) (ix(i), i = 6, 8)
  PRINT *, "isize = ", isize, "  ix(1:10) = ", ix
```

```
  ! Write another 10 elements to unit 7. File was positioned after
  ! the 8th value, so there will now be 18 values.
  WRITE(7, '(10I5)') (-i, i = 1, n)

  ! Implicit ADVANCE='YES' in WRITE, so file now positioned at
  ! beginning of 2nd record. Rewind again:
  REWIND(7)

  ! Read first 10 elements of first record
  ! (should be 1, 2, 3, 4, 5, 6, 7, 8, -1, -2)
  READ(7, '(10I5)', ADVANCE = 'NO', SIZE = isize) (ix(i), i = 1, 10)
  PRINT *, "isize = ", isize, "  ix(1:10) = ", ix

  CLOSE(7)

END PROGRAM io_specifier
```


6 Intrinsic Functions

This chapter lists the source code of the following programs:

- **char_int.f90** This program shows the use of elemental intrinsics for characters (ADJUSTL, ADJUSTR, INDEX, LEN_TRIM, SCAN, VERIFY).
- **convert.f90** This program shows the use of elemental numeric intrinsics for converting real in different ways (AINT, ANINT, INT, NINT, CELLING, FLOOR).
- **mod_int.f90** This program shows the use of elemental numeric intrinsics that do not convert (MOD, MODULO).
- **model.f90** This program shows the use of intrinsics for floating point manipulation and inquiry.

6.1 char_int.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    ch_elemental
!
!  Purpose:
!
!    This program shows the use of elemental intrinsics for
!    characters (ADJUSTL, ADJUSTR, INDEX, LEN_TRIM, SCAN, VERIFY).
!
!-------------------------------------------------------------------------------

PROGRAM ch_elemental

  IMPLICIT NONE
  CHARACTER (LEN = 2) :: chstars = '**'
  CHARACTER (LEN = 40) :: &
    char = '     5 Blanks,  30 chars,  5 blanks     '

  WRITE (*,*) 'Original : ', chstars, char, chstars
  WRITE (*,*) 'Adjustl  : ', chstars, ADJUSTL (char), chstars
  WRITE (*,*) 'Adjustr  : ', chstars, ADJUSTR (char), chstars
  WRITE (*,*) 'Index    : ', INDEX(char,'30')
  WRITE (*,*) 'Len_trim : ', LEN_TRIM (char)
  WRITE (*,*) 'Scan     : ', SCAN (char, 'cha')
  WRITE (*,*) 'Verify   : ', VERIFY (char, 'blank')

END PROGRAM ch_elemental
```


6.2 convert.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    convert
!
!  Purpose:
!
!    This program shows the use of elemental numeric intrinsics for
!    converting real in different ways (AINT, ANINT, INT, NINT,
!    CELLING, FLOOR).
!
!-------------------------------------------------------------------------------

PROGRAM convert

  IMPLICIT NONE

  INTEGER, PARAMETER :: ik = SELECTED_REAL_KIND(10)
  REAL(ik) :: pi = -3.141592653589793_ik

  WRITE(*,*) 'Kind value for reals with > 10 d.p.: ', ik
  WRITE(*,*)
  WRITE(*,*) 'Original value: ', pi
  WRITE(*,*)

  ! FORTRAN 77:

  WRITE(*,*) 'AINT    - Truncate to real    : ', AINT(pi)
  WRITE(*,*) 'ANINT   - Nearest real        : ', ANINT(pi)
  WRITE(*,*) 'INT     - Truncate to integer : ', INT(pi)
  WRITE(*,*) 'NINT    - Nearest integer     : ', NINT(pi)

  ! Fortran 90:

  WRITE(*,*) 'CEILING - Next larger integer : ', CEILING(pi)
  WRITE(*,*) 'FLOOR   - Next smaller integer: ', FLOOR(pi)

END PROGRAM convert
```

6.3 mod_int.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    noconvert
!
!  Purpose:
!
!    This program shows the use of elemental numeric intrinsics that
!    do not convert (MOD, MODULO)
!
!-------------------------------------------------------------------------------


PROGRAM noconvert

  IMPLICIT NONE

  INTEGER :: a = -100, p = 7

  WRITE(*,*) 'Original values of a and p: ', a, p
  WRITE(*,*)

  ! FORTRAN 77:

  WRITE(*,*) 'MOD     - Remainder of a modulo p: ', MOD(a,p)

  ! Fortran 90:

  WRITE(*,*) 'MODULO  - a modulo p             : ', MODULO(a,p)

END PROGRAM noconvert
```


6.4 model.f90

```
!-------------------------------------------------------------------------------
!
!  Program name:
!
!    model_numbers
!
!  Purpose:
!
!    This program shows the use of intrinsics for floating point
!    manipulation and inquiry.
!
!-------------------------------------------------------------------------------

PROGRAM model_numbers

  IMPLICIT NONE

  INTEGER, PARAMETER :: ik = SELECTED_REAL_KIND(10)
  REAL(ik), PARAMETER, DIMENSION(2) :: a=(/ 1.0_ik,3.14159265358979323846_ik /)
  REAL(ik) :: pi, one = a(1)
  INTEGER :: i, n = 3

  DO i = 1, 2
    pi = a(i)

    ! Using Intrinsics for floating point manipulation:

    WRITE (*,*)
    WRITE (*,*) 'Original value: ', pi
    WRITE (*,*)
    WRITE (*,*) 'Exponent      : ', EXPONENT (pi)
    WRITE (*,*) 'Fraction      : ', FRACTION (pi)
    WRITE (*,*) 'Nearest above : ', NEAREST (pi,one)
    WRITE (*,*) 'Rrspacing     : ', RRSPACING (pi)
    WRITE (*,*) 'Scale         : ', SCALE (pi,n)
    WRITE (*,*) 'Set_exponent  : ', SET_EXPONENT (pi,n)
    WRITE (*,*) 'Spacing       : ', SPACING (pi)
    WRITE (*,*)

    ! Using numeric enquiry intrinsics.

    WRITE (*,*) 'Digits        : ', DIGITS(pi)
    WRITE (*,*) 'Epsilon       : ', EPSILON(pi)
    WRITE (*,*) 'Huge          : ', HUGE(pi)
    WRITE (*,*) 'Max exponent  : ', MAXEXPONENT(pi)
    WRITE (*,*) 'Min exponent  : ', MINEXPONENT(pi)
    WRITE (*,*) 'Precision     : ', PRECISION(pi)
    WRITE (*,*) 'Radix         : ', RADIX(pi)
```

```
    WRITE (*,*) 'Tiny          : ', TINY(pi)
    WRITE (*,*)

END DO

END PROGRAM model_numbers
```