



# Fortran 95, Answers of Exercises

*Introductory course, 8-9 Oct 2009*

Raimo Uusvuori, Tommi Bergman, Jarmo Pirhonen, Sami Saarinen

`firstname.lastname[at]csc.fi`

CSC - IT Center for Science Ltd.

CSC - Tieteen tietotekniikan keskus Oy

Espoo, Finland

CSC – Tieteen tietotekniikan keskus Oy  
CSC – IT Center for Science Ltd.



This slide is intentionally left empty.

# Exercise 1



1. In the following the all text after the exclamation mark is interpreted as comment. Only A and B will get values.

```
A = 0.0 ; B = 370 ! Initialization ; C = 17.0 ; D = 33.0
```

2. The following is not syntactically correct Fortran 90 code:

```
Y = SIN(MAX(X1,X2)) * EXP(-COS(X3)**I) - TAN(AT&  
& AN(X4))
```

In this case the space character is not allowed after the continuation line mark on the second line, because with it ATAN(X4) is read as AT AN(X4). Therefore it is a syntax error.

3. Declaration

```
REAL :: real
```

is formally correct, but because after this redefinition and declaration the function REAL cannot be used in its original meaning. Not recommended!

## 4. Errors

- the continuation line mark & is missing from the end of the first line
- the second statement is correct, the result is the figure  $-5.6$
- the third statement is correct, it declares a character string, not logical entities
- the hyphen is not allowed in the names of the variables, should be `low_limit` or `lowlimit`
- the mantissa of the exponent notation is missing from the last statement, should be, e.g.,  $y = 1E6$

# Exercise 2

1. The following statements are not according Fortran syntax:

```
DOUBLE :: x
CHARACTER(LEN=*), PARAMETER :: "Name"
REAL x = 1.0
```

Note also, that the result of the division  $22/7$  is truncated so that the result is rounded to an integer closer to zero (integer division). Because  $22/7 \approx 3.14$  rounding is done to 3 not to 4.

2. Declaration:

```
PROGRAM constant
  IMPLICIT NONE
  REAL, PARAMETER :: k = 0.75
  WRITE (*,*) 'k = ', k
END PROGRAM constant
```

### 3. Program:

```
PROGRAM kindtest
  IMPLICIT NONE

  INTEGER :: i, int_kind, p, r, real_kind

  DO i = 1, 50
    int_kind = SELECTED_INT_KIND(i)
    WRITE (*,*) 'i = ', i, ' kind = ', int_kind
  END DO

  DO p = 1, 40
    DO r = 1, 101, 10
      real_kind = SELECTED_REAL_KIND(p,r)
      WRITE (*,*) 'p = ', p, 'r = ', r, ' kind = ', real_kind
    END DO
  END DO

END PROGRAM kindtest
```

shows that the f95 GCC (GNU) compiler gfortran and PGI compiler pgf90

- can represent integers within the range  $-10^{38} < I < 10^{38}$  and  $-10^{18} < I < 10^{18}$ , respectively; `int_kind` returns -1 outside these ranges.
- can represent real numbers with decimal precision of at least 18 and 15 digits, respectively, regardless of a decimal exponent range of 1 . . . 101 here; `real_kind` returns -1 outside these ranges. NB: Redirect output to a file, e.g.: `./a.out > log`

#### 4. Output with different precisions (the previous exercise showed that the highest precisions known by GCC and PGI compilers were 18 and 15 digits, respectively)

```
PROGRAM rprecision
  IMPLICIT NONE
  INTEGER, PARAMETER :: rkind = SELECTED_REAL_KIND(P=6)
!  INTEGER, PARAMETER :: rkind = SELECTED_REAL_KIND(P=12)
!  INTEGER, PARAMETER :: rkind = SELECTED_REAL_KIND(P=15)
!  INTEGER, PARAMETER :: rkind = SELECTED_REAL_KIND(P=18)
!  INTEGER, PARAMETER :: rkind = SELECTED_REAL_KIND(P=24)
!  INTEGER, PARAMETER :: rkind = SELECTED_REAL_KIND(P=33)
  REAL(KIND=rkind) :: x
  x = LOG(1.0_rkind + SQRT(2.0_rkind))/3 + &
    (2.0_rkind + SQRT(2.0_rkind))/15
  WRITE (*,*) PRECISION(x), x
END PROGRAM rprecision
```

# Exercise 3

1. The program prints:

```
12.000 hello  
24.000 hi  
36.000 hi hi
```

Character string constants were used in the program with different ways for giving format codes. Declaration of character string constants is easy but their values cannot be changed afterwards. On the other hand, initialization of character string variables is more tedious.

## 2. Here we have few alternatives:

```
WRITE(*,'(6F8.4)') a ! prints a 6 x 10 matrix
```

```
! reversed order for mathematical matrices
```

```
WRITE(*,'(F8.4)') a ! prints a column vector
```

```
WRITE(*,'(4I5)') H ! prints a 6 x 4 matrix
```

```
WRITE(*,'(A)') MARRAY ! words below each other
```

```
WRITE(*,'(3A10)') MARRAY ! 2 rows, 3 words per row
```

```
WRITE(*,'(L7)') CONDITION ! truth values below each other
```

```
WRITE(*,'(F8.4,F8.4)') Z ! complex numbers below each other
```

```
WRITE(*,'(F8.4,F8.4,5X,F8.4,F8.4)') Z ! two alongside
```

```
WRITE(*,'(3F8.4,3F8.4)') Z ! three alongside
```

```
WRITE(*,'(6F8.4)') Z ! three alongside
```

### 3. Most straightforward:

```
READ(11,*) a
READ(12,*) h
READ(13,*) marray
READ(14,*) condition
READ(15,*) z
```

The file `fort.11` must contain 60 real numbers, e.g., below each other or six lines each containing ten numbers. The `fort.12` contains 24 integers. Depending on the values of the character string array `marray` the file `fort.13` may contain character strings as such or separated with apostrophes or double quotation marks. Apostrophes must be used if an element contains, among other things, a space mark, a comma, a slash or if it continues to more than one line.

This slide is intentionally left empty.

# Exercise 4

## 1. Behavior of loops:

	count	values	after the i loop
DO i = 1, 5	5	1,2,3,4,5	6
DO i = 5, 0, -1	6	5,4,3,2,1,0	-1
DO i = 10, 1, -2	5	10,8,6,4,2	0
DO i = 0, 30, 7	5	0,7,14,21,28	35
DO i = 3, 2, 1	0	-	3

2. Write a `SELECT CASE` structure, which does different operations when an integer variable is negative, it is zero, or it is one of the prime numbers (3, 5, 7, 11, 13). In other cases nothing is done.

```
PROGRAM FORMAT  
  
  INTEGER :: n  
  
  DO n = -1,14  
  
    WRITE (*,*) 'n =', n  
  
    SELECT CASE(n)  
  
    CASE(:-1)  
  
      WRITE (*,*) 'Operation 1'  
  
    CASE(0)  
  
      WRITE (*,*) 'Operation 2'  
  
    CASE(3,5,7,11,13)  
  
      WRITE (*,*) 'Operation 3'  
  
    END SELECT  
  
  END DO  
  
END PROGRAM FORMAT
```

3. Write the DO loop, which sums the square roots of 100 given numbers, excluding negative numbers, and stops summing if the given number is zero. Make two versions so that one uses a CYCLE statement whereas another one does not use it.

Without CYCLE statement:

```
rsum = 0.0
DO i = 1, 100
  IF (x(i) == 0.0) THEN
    EXIT
  ELSE IF (x(i) > 0.0) THEN
    rsum = rsum + SQRT(x(i))
  END IF
END DO
```

With CYCLE statement:

```
rsum = 0.0
DO i = 1, 100
  IF (x(i) == 0.0) THEN
    EXIT
  ELSE IF (x(i) < 0.0) THEN
    CYCLE
  END IF
  rsum = rsum + SQRT(x(i))
END DO
```

The DO loop can be replaced with the following statement:

```
rsum=SUM(SQRT(x), x>0.0)
```

#### 4. Inches to millimeters. Two different output ways:

```
PROGRAM transform
  IMPLICIT NONE

  INTEGER :: i
  REAL :: inches, millimeters

  DO i = 0, 24
    inches = 0.50*i
    millimeters = 25.4*inches
    WRITE (*,'(2(F7.2))') inches, millimeters
  END DO

  WRITE (*,'(2(F7.2))') &
    (0.5*i*(/ 1.0, 25.4 /), i = 0, 24)
END PROGRAM transform
```

# Exercise 5

We write solutions of this exercise using internal procedures, which are called from the main program. Please note, that Fortran-90/95 standard does not allow an `INTERFACE` statement (nor it is needed) for internal procedures, because their interface is known.

1. Requested subprograms are `XY_POLAR` and `POLAR_XY`.

See next pages.

```
PROGRAM Coordinatetransform
IMPLICIT NONE

REAL :: X, Y, R, PHI

X = 1.0
Y = -1.0
CALL XY_Polar X,Y,R,PHI)

WRITE(*, '(A4,X,F10.6,3X,A4,X,F10.6)') 'X:',X, 'Y:',Y
WRITE(*, '(A4,X,F10.6,3X,A4,X,F10.6)') 'R:',R, 'PHI:',PHI
WRITE(*,*)' '

R = 1.0
PHI = 1.04719755119660

CALL Polar_XY(R,PHI,X,Y)

WRITE(*, '(A4,X,F10.6,3X,A4,X,F10.6)') 'R:',R, 'PHI:',PHI
WRITE(*, '(A4,X,F10.6,3X,A4,X,F10.6)') 'X:',X, 'Y:',Y
```

CONTAINS

```
SUBROUTINE Polar_XY(R,PHI,X,Y)
```

```
  IMPLICIT NONE
```

```
  REAL,INTENT(IN) :: R, PHI
```

```
  REAL,INTENT(OUT) :: X, Y
```

```
  X = R*COS(PHI)
```

```
  Y = R*SIN(PHI)
```

```
END SUBROUTINE Polar_XY
```

```
SUBROUTINE XY_Polar(X,Y,R,PHI)
```

```
  IMPLICIT NONE
```

```
  REAL,INTENT(IN) :: X, Y
```

```
  REAL,INTENT(OUT) :: R, PHI
```

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

```
  PHI = ATAN2(Y,X)
```

```
END SUBROUTINE XY_POLAR
```

```
END PROGRAM Coordinatetransform
```

2. A trouble here is the concept of character string: if the character string is as an input, where it ends? If we want to reserve the possibility, e.g., to include spaces in the input, the input character string must be enclosed in apostrophes or double quotation marks.

```
PROGRAM verif
  IMPLICIT NONE

  CHARACTER (LEN=80):: mstring
  INTEGER:: out

  WRITE(*,*) 'Give a character string'
  READ(*,*) mstring
! IF A SPACE MARK IS WANTED TO BE INCLUDED IN THE CHARACTER
! STRING ,APOTROPHES MUST BE USED

  IF (test(TRIM(mstring))) THEN
    WRITE(*,*) 'Only numerals'
  ELSE
    WRITE(*,*) 'Character string contains also other things than numerals'
  END IF
```

```
CONTAINS
FUNCTION test(string) RESULT(res)
  IMPLICIT NONE

  CHARACTER (LEN=*):: string
  LOGICAL:: res

  IF (VERIFY(string,'0123456789')==0) THEN
    res=.true.
  ELSE
    res=.false.
  END IF
  RETURN
END FUNCTION test
END PROGRAM verif
```

3. We can use either the standard function `LOG10` (common logarithm) or `LOG` (natural logarithm). Note, that in Fortran 90/95 the previous allows only a positive real argument but the latter also a complex variable! In the following solution we stay in real number area and use the standard function for common logarithm.

```
PROGRAM logarithm
  IMPLICIT NONE

  REAL :: b, x, y

  WRITE(*,*) 'Give base and argument for logarithm'
  READ(*,*) b, x

  IF ( (b>0) .AND. (x>0) ) THEN
    WRITE(*,*) 'Value of logarithm:', LOGB(b,x)
  ELSE
    WRITE(*,*) 'Both numbers must be positive.'
  END IF

  CONTAINS
  REAL FUNCTION LOGB(b,x)
    IMPLICIT NONE

    REAL :: b, x
    LOGB = LOG10(x)/LOG10(b)
  END FUNCTION LOGB
END PROGRAM logarithm
```

## 4. Calculating the greatest common factor:

```
PROGRAM gcf_calc
! Calculating the greatest common factor of integers m and n
! using the Euklidian algorithm.
  IMPLICIT NONE

  INTEGER, PARAMETER :: int_kind = SELECTED_INT_KIND(9)
  INTEGER (KIND=int_kind) :: m, n

  WRITE (*,*) 'Give positive integers m and n:'
  READ (*,*) m, n
  WRITE (*,*) 'm:', m, ' n:', n
  WRITE (*,*) ' The greatest common factor:', gcf(m,n)

  CONTAINS
  RECURSIVE FUNCTION gcf(m,n) RESULT(gcf_value)
    INTEGER(KIND=int_kind), INTENT(IN) :: m, n
    INTEGER(KIND=int_kind) :: gcf_value

    IF (n == 0) THEN
      gcf_value = m
    ELSE
      gcf_value = gcf(n, MOD(m,n))
    END IF
  END FUNCTION gcf
END PROGRAM gcf_calc
```



This slide is intentionally left empty.

## 5. Calculating "Tribonacci numbers":

```
PROGRAM trib
  IMPLICIT NONE

  INTEGER :: n, m

  n = 1
  DO
    WRITE(*,'(A)',ADVANCE='no') 'Give n: '
    READ(*,*) n
    IF (n <= 0) EXIT
    m = tribonacci(n)
    WRITE(*,*) 'n =', n, ', m =', m
  END DO

  CONTAINS
  RECURSIVE FUNCTION tribonacci(n) RESULT(res)
    IMPLICIT NONE

    INTEGER, INTENT(IN) :: n
    INTEGER :: res

    IF (n <= 3) THEN
      res = 1
    ELSE
      res = tribonacci(n-1) + &
        tribonacci(n-2) + tribonacci(n-3)
    END IF
  END FUNCTION tribonacci
END PROGRAM trib
```

Result:

```
Give n: 15  
n = 15 , m = 2209  
Give n: -1
```

Performance of the routine is very poor when  $n$  increases — try, for example,  $n = 20$ ,  $n = 25$  and  $n = 30$ . You can add the `SAVE` variable to the function. It calculates, how many times the function is called (value can be printed, e.g., with negative argument).

## 6. Let us declare and test function degrees:

```
PROGRAM degreetest
  IMPLICIT NONE

  INTRINSIC ASIN, ACOS, ATAN

  WRITE (*,*) 'arcsin(0.5): ', degrees(ASIN,0.5)
  WRITE (*,*) 'arccos(0.5): ', degrees(ACOS,0.5)
  WRITE (*,*) 'arctan(1.0): ', degrees(ATAN,1.0)

  CONTAINS
  REAL FUNCTION degrees(f, x)
    IMPLICIT NONE

    REAL, EXTERNAL :: f
    REAL :: x
    INTRINSIC ATAN

    degrees = 45*f(x)/ATAN(1.0)
  END FUNCTION degrees
END PROGRAM degreetest
```

# Compilation of degreetest

Let's try to compile with GCC gfortran (here f90 → gfortran 4.1.2).

```
% f90 -o degreetest degreetest.f90
```

```
In file degreetest.f90:6
```

```
WRITE (*,*) 'arcsin(0.5): ', degrees(ASIN,0.5)
                                     1
```

```
Error: Type/rank mismatch in argument 'f' at (1)
```

```
In file degreetest.f90:7
```

```
WRITE (*,*) 'arccos(0.5): ', degrees(ACOS,0.5)
                                     1
```

```
Error: Type/rank mismatch in argument 'f' at (1)
```

```
In file degreetest.f90:8
```

```
WRITE (*,*) 'arctan(1.0): ', degrees(ATAN,1.0)
                                     1
```

```
Error: Type/rank mismatch in argument 'f' at (1)
```

gfortran 4.2.4 behaves similarly but prints also a column number on a line.

What's wrong? There is a sort of an error in syntax. Declaration of intrinsic functions is not complete in the main program. Try add type `REAL` for declarations of intrinsic functions. This is according to the syntax. When you have now comma separated list of specifications you need also double colon in the declaration:

```
REAL, INTRINSIC :: ASIN, ACOS, ATAN
```

Now GCC f95 or f90 compilers 4.1.2 and 4.2.4 compile the program `degreetest` and it prints:

```
arcsin(0.5):      30.00000  
arccos(0.5):      60.00000  
arctan(1.0):      45.00000
```

GCC gfortran 4.3.2 and PGI compilers `pgf90` and `pgf95` don't care anything about this `REAL` statement. They compile also the uncorrected original program.

# Exercise 6

1. In one of the lectures concerning Arguments of a procedure there was introduced the routine `FUNCTION gauss` for integration of scalar function  $f(x)$  using the Gauss integration formula

$$\int_a^b f(x)dx \approx \frac{b-a}{2} \left[ f\left(\frac{a+b}{2} - \frac{b-a}{2\sqrt{3}}\right) + f\left(\frac{a+b}{2} + \frac{b-a}{2\sqrt{3}}\right) \right]$$

where the integration interval is  $[a, b]$ . Add optional arguments to the program so that

- if the lower integration limit  $a$  is not given, it is set to  $a = 0$ ,
- if the upper integration limit  $b$  is not given, it is set to  $b = a + 1$ .

In the following example (`integrate_SIN.f90`) integration is done over standard function `SIN`.

```
PROGRAM integrate
  IMPLICIT NONE
  REAL :: a, b
  ! GCC f90/f95/gfortran requires declaration
  ! to REAL for intrinsic function
  REAL, INTRINSIC :: SIN
  a = 1.d0
  b = 2.d0
  WRITE (*,*) gauss(SIN, a, b)

CONTAINS
  REAL FUNCTION gauss(f,a,b)
    IMPLICIT NONE
    REAL :: f
    ! GCC f90/f95/gfortran requires declaration
    ! to REAL for intrinsic function f,
    ! but it does not accept INTERFACE block for it
```

```
REAL, INTENT(IN), OPTIONAL :: a, b
REAL :: at, bt
REAL :: p1, p2
IF (PRESENT(a)) THEN
    at = a
ELSE
    at = 0.0
END IF
IF (PRESENT(b)) THEN
    bt = b
ELSE
    bt = at + 1.0
END IF
p1 = 0.5*(at+bt) - (bt-at)/(2.0*SQRT(3.0))
p2 = 0.5*(at+bt) + (bt-at)/(2.0*SQRT(3.0))
gauss = 0.5*(bt-at)*(f(p1) + f(p2))
END FUNCTION gauss
END PROGRAM integrate
```

2. We define now ourselves an integrand function, e.g.,  $f(x) = x^2 + x$ . An elegant solution would be to put the routine inside a module. However, we write it here as an external routine.

```
REAL FUNCTION Integrand(x)
  IMPLICIT NONE
  REAL, INTENT(IN) :: x

  Integrand = x*x + x

END FUNCTION Integrand
```

The `INTERFACE` block for the integrand function is added inside the function `gauss` and declaration of `f` to `REAL` is removed.

```
INTERFACE
REAL FUNCTION f(x)
  REAL :: x
  END FUNCTION f
END INTERFACE
```

In the main program the declaration `REAL, INTRINSIC :: SIN` is replaced with the declaration `REAL, EXTERNAL :: Integrand` and the call to the function is replaced by `gauss(Integrand, a, b)`. The program (`integrate_Integrand.f90`) looks now like this:

```
PROGRAM integrate
  IMPLICIT NONE
  REAL :: a, b
  REAL, EXTERNAL :: Integrand
  a = 1.d0
  b = 2.d0
  WRITE (*,*) gauss(Integrand, a, b)
```

```
CONTAINS
  REAL FUNCTION gauss(f,a,b)
    IMPLICIT NONE

    INTERFACE
      REAL FUNCTION f(x)
        REAL :: x
      END FUNCTION f
    END INTERFACE

    REAL, INTENT(IN), OPTIONAL :: a, b
    REAL :: at, bt
    REAL :: p1, p2

    IF (PRESENT(a)) THEN
      at = a
    ELSE
      at = 0.0
    END IF
```

```
IF (PRESENT(b)) THEN
  bt = b
ELSE
  bt = at + 1.0
END IF

p1 = 0.5*(at+bt) - (bt-at)/(2.0*SQRT(3.0))
p2 = 0.5*(at+bt) + (bt-at)/(2.0*SQRT(3.0))
gauss = 0.5*(bt-at)*(f(p1) + f(p2))
END FUNCTION gauss
END PROGRAM integrate
```

The program is compiled by using the command (assuming that the names of the source files are as follows):

```
% f90 -o integrate integrand.f90 integrate_Integrand.f90
```

3. *Additional exercise.* Extend the routine you declared above so that the user can define desired number of integration points using an optional argument.

*Hint:* Find out suitable integration formulas from literature. In Finnish there is such like CSC's Guide for Numeric methods: *Numeeriset menetelmät*. Look at also the file `$DOC/f90/examples/u_integrate.f90`, where an adaptive integration routine is defined.

# Exercise 7

1. A comma is missing from the declaration after `REAL` attribute:

```
REAL, DIMENSION(1:3,2:3) :: aa
```

2. (11.2) Declare the integer array `iarray`, which contains 3 rows and 4 columns. Initialize the first row of the array with integers 1-4 in this order from left to right, the second line with integers 5-8, and the lowest line putting integer -2 to every column. Print `iarray` row by row – so that each output line contains the elements of one row of the array.
3. Build also the  $3 \times 8$  integer array `bigarray`, where the first 4 columns are identical with the array `iarray` in the previous exercise, and the 4 last columns are obtained from the columns of the array `iarray` by multiplying them with the number 3 and adding 5 to the product. Use array syntax in statements.

4. Write a subprogram which can do automatically the operation described in the previous exercise. The smaller array is given to the subprogram as input and it must return the double sized array, whose elements are constructed as described above. The following program code contains solutions to sub-exercises 2-4.

```
PROGRAM array2
  IMPLICIT NONE
  INTEGER, DIMENSION(3,4) :: iarray
  INTEGER, DIMENSION(3,8) :: bigarray,bigarray1, &
                                bigarray2,bigarray3

  INTEGER :: i,j

  iarray(1,:)=(/ (i,i=1,4) /)
  iarray(2,:)=(/ (i,i=5,8) /)
  iarray(3,:)=-2
```

```
WRITE(*,'(4I5)') ((iarray(i,j),j=1,4),i=1,3)

bigarray(:,1:4)=iarray
bigarray(:,5:8)=3*iarray+5

WRITE(*,'(8I4)') ((bigarray(i,j),j=1,8),i=1,3)

! let's try first assumed shaped arrays
CALL widearray1(iarray,bigarray1,3,4)
WRITE(*,'(8I4)') ((bigarray1(i,j),j=1,8),i=1,3)

! the explicit shape arrays
CALL widearray2(iarray,bigarray2,3,4)
WRITE(*,'(8I4)') ((bigarray2(i,j),j=1,8),i=1,3)

! and finally array-valued function
bigarray3=widearray3(iarray)
WRITE(*,'(8I4)') ((bigarray3(i,j),j=1,8),i=1,3)
```

CONTAINS

```
SUBROUTINE widearray1(iarray,bigarray,m,n)
  IMPLICIT NONE
  INTEGER, DIMENSION(:, :) :: iarray,bigarray
  INTEGER :: m,n

  bigarray(:,1:n)=iarray
  bigarray(:,n+1:2*n)=3*iarray+5
END SUBROUTINE widearray1
```

```
SUBROUTINE widearray2(iarray,bigarray,m,n)
  IMPLICIT NONE
  INTEGER :: m,n
  INTEGER, DIMENSION(m,n) :: iarray
  INTEGER, DIMENSION(m,2*n) :: bigarray

  bigarray(:,1:n)=iarray
  bigarray(:,n+1:2*n)=3*iarray+5
END SUBROUTINE widearray2
```

```
FUNCTION widearray3(iarray) RESULT(bigarray)
  IMPLICIT NONE
  INTEGER, DIMENSION(:, :) :: iarray
  INTEGER, DIMENSION(SIZE(iarray,1), 2*SIZE(iarray,2)) &
    :: bigarray
  INTEGER n

  n=SIZE(iarray,2)
  bigarray(:,1:n)=iarray
  bigarray(:,n+1:2*n)=3*iarray+5

END FUNCTION widearray3
END PROGRAM array2
```

5. Declare two pointer variables. Set one of them to point to the (statically allocated) one-dimensional array of real numbers and another one to the sixth element of the same array.

```
REAL, DIMENSION(:), POINTER :: p1, p2
REAL, POINTER :: p3
REAL, DIMENSION(10), TARGET :: x
p1 => x
p2 => x(6:6)
p3 => x(6)
```

There are thus two different ways to point to a certain element of one dimensional array.

6. Use the pointer variable to assign odd numbered elements of an array of the value 1.0 and even numbered elements of the value -1.0.

```
PROGRAM point
  IMPLICIT NONE
  REAL, DIMENSION(:), POINTER :: p1, p2
  REAL, DIMENSION(10), TARGET :: x

  p1 => x(2:SIZE(x):2)
  p2 => x(1:SIZE(x):2)
  p1 = -1.0
  p2 = 1.0
  WRITE (*,*) x
END PROGRAM point
```

7. Make a subprogram which returns the array of the desired size in a pointer variable. Give the size of the array as an argument for the subprogram. The array is allocated in a subprogram.

```
PROGRAM reservation
  IMPLICIT NONE
  REAL, DIMENSION(:), POINTER :: rarray
  call reserve(rarray, 100)
  WRITE (*,*) SIZE(rarray)
CONTAINS
  SUBROUTINE reserve(p,n)
    IMPLICIT NONE
    REAL, DIMENSION(:), POINTER :: p
    INTEGER, INTENT(IN) :: n
    INTEGER :: allocstat
    ALLOCATE(p(n), STAT=allocstat)
    IF (allocstat /= 0) THEN
      STOP 'Allocation of memory failed!'
    END IF
  END SUBROUTINE reserve
END PROGRAM reservation
```

Please note, that the actual and dummy argument must be of the same type `POINTER`. `ALLOCATABLE` cannot be used, even though the pointer variable is used only as a dynamically allocating array.

# Exercise 8

## Modified code:

```
PROGRAM table1
  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 10
  REAL, DIMENSION(n) :: a, b
  REAL, DIMENSION(n,n) :: c, d
  INTEGER :: j
  CALL RANDOM_NUMBER(b)
  CALL RANDOM_NUMBER(c)
  WRITE (*,*) b
  d = 1.0 - c
  a(2:n) = b(2:n) - b(1:n-1)
  DO j = 2, n
    c(2:n,j) = a(j) * c(1:n-1,j) + d(2:n,j)
  END DO
  WRITE (*,*) 'answer = ', c(n,n)
END PROGRAM table1
```

# Exercise 9



## Modified code:

```
PROGRAM table2
  IMPLICIT NONE
  REAL, DIMENSION(100,100) :: x
  REAL, DIMENSION(100) :: y, z, s
  REAL :: a
  INTEGER, DIMENSION(1) :: k
  INTEGER :: i, j

  y = 0.1* (/ (j, j = 1, 100) /)

  DO i = 1, 100
    x(i,:) = 1.0 + y
  END DO

  DO j = 1, 100
    s(j) = SUM(x(:,j))
  END DO

  z = SQRT(y)/s
  WHERE (z < 0.001) z = 0.0
  a = MAXVAL(z)
  k = MAXLOC(z)

  WRITE (*,*) a, k
END PROGRAM table2
```

# Exercise 10

1. The case  $n=0$  must be handled as a special case. Other cases use the same formula.

```
PROGRAM ROOTS
  COMPLEX, PARAMETER :: I=(0.0,1.0)
  REAL, PARAMETER :: PI=3.1415926
  COMPLEX :: Z
  INTEGER N,K,NIN

  WRITE(*,*) 'Give an integer N'
  READ(*,*) NIN
  WRITE(*, '(A4,I2,/)') 'N = ', NIN

  N=ABS(NIN)
  IF (N == 0) THEN
    WRITE(*,*) 'Equation X**0=1 has infinite solutions'
    STOP
  ENDIF
```

```
WRITE(*,'(A30,I2,A8,/)') &
  'Solutions of the equation X**(',NIN,')=1 are:'
WRITE(*,'(3X,A5,8X,A5)') 'RE(X)', 'IM(X)'

DO K=0,N-1
  Z=EXP(2*PI*K*I/N)
  WRITE(*,'(F9.6,4X,F9.6)') Z
END DO
END PROGRAM ROOTS
```

2. The elements of vectors V1 and V2 are assigned using an array constructor.

```
PROGRAM DOTPRODUCT

    IMPLICIT NONE
    REAL :: DP
    REAL, DIMENSION(5) :: V1, V2

    V1=( / 1.0, 2.0, 3.0, 4.0, 5.0 / )
    V2=( / -3.0, 5.0, -2.0, 1.0, -3.0 / )

    DP=DOT_PRODUCT(V1, V2)

    WRITE(*,*)V1
    WRITE(*,*)V2
    WRITE(*,*)DP

END PROGRAM DOTPRODUCT
```

3. The elements of the matrices are given row by row (row-major order as in C) using array constructors. Please note, that output does not occur automatically row by row, because matrices are saved in Fortran in column-major order.

```
PROGRAM MATRIXPRODUCT

  IMPLICIT NONE

  REAL, DIMENSION( 5, 4) :: A
  REAL, DIMENSION( 4, 3) :: B
  REAL, DIMENSION( 5, 3) :: MP

  REAL, DIMENSION( 4, 5) :: AT
  REAL, DIMENSION( 3, 4) :: BT
  REAL, DIMENSION( 3, 5) :: MPT

  INTEGER I
```

```
A(1,:)=(/ 1.0,2.0,3.0,4.0/)
A(2,:)=(/ -3.0,5.0,-2.0,1.0/)
A(3,:)=(/ 1.0,0.0,0.0,4.0/)
A(4,:)=(/ 2.0,2.0,2.0,0.0/)
A(5,:)=(/ 1.0,0.0,3.0,4.0/)
```

```
B(1,:)=(/ 0.0,5.0,-1.0/)
B(2,:)=(/ 2.0,0.0,-2.0/)
B(3,:)=(/ -3.0,2.0,4.0/)
B(4,:)=(/ 1.0,2.0,3.0/)
```

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

```
AT=TRANSPPOSE(A)
```

```
BT=TRANSPPOSE(B)
```

```
MPT=MATMUL(BT,AT)
```

```
DO I=1,5
  WRITE(*,'(F8.4,3X,F8.4,3X,F8.4)') MP(I,:)
END DO

WRITE(*,*)' '

DO I=1,3
  WRITE(*,'(F8.4,3X,F8.4,3X,F8.4,3X,F8.4,3X,F8.4)') &
    MPT(I,:)
END DO
END PROGRAM MATRIXPRODUCT
```

4. The program could be improved so that it would tell, which number is in question when such is found from the character string.

```
PROGRAM NUMSCAN
```

```
    IMPLICIT NONE
```

```
    CHARACTER(LEN=80) :: CHARSIN
```

```
    INTEGER :: LOCOUT
```

```
    WRITE(*,*) 'Give a character string'
```

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

```
    LOCOUT=SCAN(CHARSIN,'0123456789')
```

```
    WRITE(*,*) ' '
```

```
    IF (LOCOUT > 0) THEN
```

```
        WRITE(*,'(A28,X,I2)') 'The first number in location',LOCOUT
```

```
    ELSE
```

```
        WRITE(*,*) 'No numbers among the first 80 characters'
```

```
    ENDIF
```

```
END PROGRAM NUMSCAN
```



This slide is intentionally left empty.

# Exercise 11



The following program is one solution to the Subexercises 1.-3. Please note, that the model solution is from the previous century (1900).

```
PROGRAM SYNTY
IMPLICIT NONE

INTEGER, PARAMETER :: karkea = SELECTED_INT_KIND(2)
INTEGER, PARAMETER :: tarkempi = SELECTED_INT_KIND(4)
CHARACTER (LEN=12) :: paivays
TYPE synt_aika
  CHARACTER (LEN=20) :: etunimi
  CHARACTER (LEN=20) :: sukunimi
  INTEGER (KIND=karkea) :: paiva
  INTEGER (KIND=karkea) :: kuukausi
  INTEGER (KIND=tarkempi) :: vuosi
END TYPE synt_aika

TYPE(synt_aika) :: tieto

tieto%vuosi=1900
tieto%kuukausi=0
tieto%paiva=0
```

```
WRITE(*,*) 'Nimi?'
!           'Name?'
READ (*,*) tieto%etunimi,tieto%sukunimi

WRITE(*,*) 'Vuosi?'
!           'Year?'
READ (*,*)tieto%vuosi
IF (tieto%vuosi.LT.0) THEN
  STOP
ENDIF

IF (tieto%vuosi.LT.100) THEN
  tieto%vuosi=tieto%vuosi+1900
ENDIF

IF ((tieto%vuosi.LT.1800).OR.(tieto%vuosi.GT.2000)) THEN
!           "I don't believe! I move to this century"
  WRITE(*,*) "En usko! Siirrän tälle vuosisadalle",tieto%vuosi
  tieto%vuosi=tieto%vuosi-100*(tieto%vuosi/100)+1900
ENDIF

DO WHILE ((tieto%kuukausi.LT.1).OR.(tieto%kuukausi.GT.12))
  WRITE(*,*) 'Kuukausi?'
!           'Month?'
  READ (*,*)tieto%kuukausi
END DO
```

```
DO WHILE ((tieto%paiva.LT.1).OR.(tieto%paiva.GT.31))
  WRITE(*,*) 'Paiva?'
  !           'Day?'
  READ (*,*)tieto%paiva
END DO

! Tulostus kahdessa osassa: ensin kootaan paivamaara
! WRITE-komennolla sisäiseen tiedostoon paivays,
! ja lopuksi tulostetaan nimi+päivämäärä ruudulle.
! Vertaile tulostuksen muotoilutapoja.
!
! Printing in two parts: first the date is collected
! to an internal file with WRITE statement and finally
! name+birthdate is printed to the screen.
! Compare different output formats.

WRITE(paivays,'("(",I2.2,".",I2.2,".",I4,")")') &
  tieto%paiva,tieto%kuukausi,tieto%vuosi

WRITE(*,'(5A)')TRIM(tieto%etunimi)," ", &
  TRIM(tieto%sukunimi)," ",paivays

END PROGRAM SYNTY
```



This slide is intentionally left empty.

## The module for the operations described in the Subexercise 4.

```
MODULE kurssimod
  IMPLICIT NONE

  INTEGER, PARAMETER :: maxlkm=20
  TYPE kurssilainen
    CHARACTER (LEN=20) :: etunimi
    CHARACTER (LEN=20) :: sukunimi
    REAL :: arvosana
  END TYPE kurssilainen

  TYPE(kurssilainen), DIMENSION(maxlkm) :: oppilaat

CONTAINS
  SUBROUTINE lueykksi(oppilaat, maara_nyt, max_maara)
    INTEGER, INTENT(IN) :: max_maara
    INTEGER, INTENT(INOUT) :: maara_nyt
    TYPE(kurssilainen), DIMENSION(max_maara) :: oppilaat

    IF (maara_nyt.GE.max_maara) THEN
      WRITE(*,*) "Kurssi taynnä"
!       "The course is fully occupied"
      GOTO 9999
    ENDIF
  END SUBROUTINE
END MODULE
```

```
WRITE(*,*)"Etu- ja sukunimi?"
!           "First and last name?"
READ(*,*) oppilaat(maara_nyt+1)%etunimi, &
           oppilaat(maara_nyt+1)%sukunimi
           maara_nyt=maara_nyt+1

9999      CONTINUE
          RETURN
END SUBROUTINE lueykxi

SUBROUTINE tulosta(oppilaat, maara_nyt, max_maara)
  INTEGER, INTENT(IN) :: max_maara,maara_nyt

  TYPE(kurssilainen), DIMENSION(max_maara) :: oppilaat
  INTEGER :: I

  DO i=1,maara_nyt
    WRITE(*,'(A,F5.1)') TRIM(oppilaat(i)%etunimi)//' '&
      TRIM(oppilaat(i)%sukunimi),oppilaat(i)%arvosana
  END DO
RETURN
END SUBROUTINE tulosta
```

```
SUBROUTINE arvostele(oppilaat, maara_nyt, max_maara)
  INTEGER, INTENT(IN) :: max_maara, maara_nyt

  TYPE(kurssilainen), DIMENSION(max_maara) :: oppilaat
  INTEGER I

  DO i=1,maara_nyt
    oppilaat(i)%arvosana=-1.0
    DO WHILE (ABS(oppilaat(i)%arvosana-2.5).GT.2.5)
!           'Give the mark for the student '
      WRITE (*,*) 'Anna arvosana oppilaalle ',&
        TRIM(oppilaat(i)%etunimi), ' ',&
        TRIM(oppilaat(i)%sukunimi)
! arvosana = mark, grade, must be between 0-5.
      READ(*,*)oppilaat(i)%arvosana
    END DO
  END DO

  RETURN
END SUBROUTINE arvostele
```

```
SUBROUTINE keskiarvo(oppilaat, maara_nyt, max_maara)
  INTEGER, INTENT(IN) :: max_maara, maara_nyt
  INTEGER I
  REAL ka

  TYPE(kurssilainen), DIMENSION(max_maara) :: oppilaat

  ka=SUM(oppilaat%arvosana)/maara_nyt
  WRITE(*,'(A,F5.2)')'Keskiarvo on',ka
!                               'The average is'

  RETURN
  END SUBROUTINE keskiarvo
END MODULE kurssimod
```

## The main program using the previous module

```
PROGRAM kurssi
  USE kurssimod
  IMPLICIT NONE
  INTEGER :: maara, ohje

! maara (= määrä) = amount, number; ohje = guide, usage

  maara=0

  DO
    WRITE(*,*) 'Valitse toiminto'
    WRITE(*,*) '1 - syötä oppilastieto'
    WRITE(*,*) '2 - arvostelee'
    WRITE(*,*) '3 - tulosta tiedot'
    WRITE(*,*) '4 - laske keskiarvo'
    WRITE(*,*) '5 - lopeta'

!           'Select'
!           '1 - give student data'
!           '2 - mark'
!           '3 - print data'
!           '4 - calculate average'
!           '5 - exit'
    READ(*,*) ohje
```

```
SELECT CASE(ohje)
  CASE(1)
    CALL lueyksi(oppilaat,maara,maxlkm)
  CASE(2)
    CALL arvosteleva(oppilaat,maara,maxlkm)
  CASE(3)
    CALL tulosta(oppilaat,maara,maxlkm)
  CASE(4)
    CALL keskiarvo(oppilaat,maara,maxlkm)
  CASE(5)
    EXIT
  CASE DEFAULT
    WRITE(*,*)'Valitse jokin annetuista vaihtoehtoista!'
!           !Select one alternative!'
  END SELECT
END DO
END PROGRAM kurssi
```

# Exercise 12



## 1. The main program and the subprogram

### a) The first trial

```
tyyppi = type
uusi_tyyppi = new_type
ali = sub
```

```
PROGRAM tyyppi
  IMPLICIT NONE
  TYPE uusi_tyyppi
    REAL :: x
  END TYPE uusi_tyyppi

  TYPE(uusi_tyyppi) :: var

  var%x=1.0
  CALL ali(var)

END PROGRAM tyyppi
```

```
SUBROUTINE ali(var)
  IMPLICIT NONE
  TYPE uusi_tyyppi
    REAL :: x
  END TYPE uusi_tyyppi

  TYPE(uusi_tyyppi) :: var

  PRINT *,var%x

END SUBROUTINE ali
```

Succeeding of compilation depends on the compiler. All available F90/F95 compilers on Hippu, GNU gfortran, PGI pgf90, PathScale pathf90 and Intel ifort, accept the program and subprogram, but some other compilers do not let pass a derived type variable to the subprogram if it is not used with USE statement.

b) The following is more correct version (the module and the main program)

```
MODULE uusi
  IMPLICIT NONE
  TYPE uusi_tyyppi
    REAL :: x
  END TYPE uusi_tyyppi

  CONTAINS

  SUBROUTINE ali(var)
    IMPLICIT NONE
    TYPE(uusi_tyyppi) :: var
    PRINT *,var%x
  END SUBROUTINE ali

END MODULE uusi
```

```
PROGRAM tyyppi
  USE uusi, ONLY: uusi_tyyppi, ali
  IMPLICIT NONE
  TYPE(uusi_tyyppi) :: var

  var%x=1.0

  CALL ali(var)

END PROGRAM tyyppi
```

## 2. The module containing the kind type parameter

tark = abbreviation of tarkka = precise

```
MODULE tark
  IMPLICIT NONE
  INTEGER, PARAMETER:: prec=SELECTED_REAL_KIND(12,100)
END MODULE tark
```

## The main program and the subprogram

summaa = sum up  
summa = sum

```
PROGRAM summaa
  USE tark, ONLY:prec
  IMPLICIT NONE
  REAL (prec) :: a,b
  REAL (prec), EXTERNAL:: summa

  a=1
  b=2
  PRINT *,summa(a,b)

END PROGRAM summaa
```

```
FUNCTION summa(a,b) RESULT(c)
  USE tark, ONLY:prec
  IMPLICIT NONE
  REAL (prec) :: a,b,c

  c=a+b

END FUNCTION summa
```

## Compilation an execution of the program:

```
hippul:~> gfortran tarkmod_ex12_2.f90 summaa_ex12_2.f90
hippul:~> ./a.out
  3.0000000000000000
```

This is the correct result.

### 3. Another addend is integer.

- a) The module `tark` and the main program `summaa` of the previous exercise are used, but the function `summa` is now:

```
FUNCTION summa(a,b) RESULT(c)
  USE tark, ONLY:prec
  IMPLICIT NONE
  REAL (prec) :: a,c
  INTEGER :: b

  c=a+b

END FUNCTION summa
```

#### Execution of the program:

```
hippul> ./a.out
1.0000000000000000
```

The result is erroneous because, the real number as argument is interpreted to be integer. The value of `b` becomes 0 in return from `summa`.

- b) The previous function `summa` is put into the module `plus` and removed from the main source file program:

```
MODULE plus
  IMPLICIT NONE

  CONTAINS

  FUNCTION summa(a,b) RESULT(c)
    USE tark, ONLY:prec
    IMPLICIT NONE
    REAL (prec) :: a,c
    INTEGER :: b

    c=a+b

  END FUNCTION summa

END MODULE plus
```

The main program is also modified as follows, but the module `tark` of the previous exercises is used unmodified:

```
PROGRAM summaa
  USE tark, ONLY:prec
  USE plus, ONLY:summa
  ...
  ! GCC gfortran or pgf90 does not accept the following declaration,
  ! because "summa" is now USE associated. So it must be removed or
  ! Commented:
  ! REAL (prec), EXTERNAL:: summa
  ! The same applies to the following kind of declarations. Either
  ! gfortran or pgf90 or both do not accept these:
  ! REAL (prec) :: summa
  ! EXTERNAL :: summa
  ...
```

Now the gfortran compiler gives the following error message:

```
hippu1~> f90 tarkmod_ex12_2.f90 plusmod_ex12_3b.f90  
summaa_ex12_3b.f90  
summaa_ex12_3b.f90:11.18:
```

```
    PRINT *,summa(a,b)  
                1
```

```
Error: Type/rank mismatch in argument 'b' at (1)
```

and pgf90/95 the following one:

```
hippu~> f90 tarkmod_ex12_2.f90 plusmod_ex12_3b.f90  
summaa_ex12_3b.f90  
tarkmod_ex12_2.f90:  
plusmod_ex12_3b.f90:  
summaa_ex12_3b.f90:  
PGF90-S-0188-Argument number 2 to summa: type mismatch  
(summaa_ex12_3b.f90: 11)  
    0 inform,    0 warnings,    1 severes, 0 fatal for summaa
```

Please, note that the module files must be before the main program file on the command line, so that modules are compiled before they are USED in the compiled main program.

4. The global `point` type. The module `tark` above is used here and in compilation.

```
MODULE point
  USE tark, ONLY: prec
  implicit none

  TYPE point
    REAL (prec):: x,y
  END TYPE point

  CONTAINS

  FUNCTION plus(p1,p2) RESULT(p)
    IMPLICIT NONE
    TYPE (point):: p1,p2,p

    p%x=p1%x+p2%x
    p%y=p1%y+p2%y

  END FUNCTION plus

END MODULE point
```

```
PROGRAM test
  USE tark, ONLY: prec
  USE point, ONLY: point, plus
  IMPLICIT NONE

  TYPE (point):: p1,p2

  p1=point(1,2)
  p2=point(3,4)

  PRINT *, plus(p1,p2)

END PROGRAM test
```

5. Object oriented programming. Private components. Module `tark` above is used here and in compilation.

a) No initialization

```
MODULE point
  USE tark, ONLY: prec
  implicit none

  TYPE point
    PRIVATE
    REAL (prec):: x,y
  END TYPE point
  ...
```

**GCC gfortran 4.1.2 compiles this without error messages, but GCC 4.2.4 and 4.2.3 and PGI 8 and 9 pgf90/pgf95 give error messages. This from gfortran 4.2.3:**

```
hippu1~> gfortran tarkmod_ex12_2.f90 point_ex12_5a.f90
point_ex12_5a.f90:30.14:
  p1=point(1,2)
                1
Error: Structure constructor for 'point' at (1) has PRIVATE
components
point_ex12_5a.f90:31.14:
  p2=point(3,4)
                1
Error: Structure constructor for 'point' at (1) has PRIVATE
components
point_ex12_5a.f90:33.22:
  PRINT *, plus(p1,p2)
                1
Error: Data transfer element at (1) cannot have PRIVATE components
```

b) Thus there is need for the subprograms for initialization and printing of "point"s:

```
SUBROUTINE init(p,x,y)
  IMPLICIT NONE
  TYPE (point):: p
  REAL (prec):: x,y

  p%x=x
  p%y=y

END SUBROUTINE init

SUBROUTINE out(p)
  IMPLICIT NONE
  TYPE (point):: p

  PRINT *, p
END SUBROUTINE out

END MODULE point
```

## The main program:

```
PROGRAM test
  USE tark, ONLY: prec
  USE point, ONLY: point, plus, init, out
  IMPLICIT NONE

  TYPE (point):: p1,p2

  CALL init(p1,1.0_prec,2.0_prec)
  CALL init(p2,3.0_prec,4.0_prec)
  CALL out(plus(p1,p2))

END PROGRAM test
```

Now also the PGI and newer GCC compilers compile this program.

# Exercise 13

## 1. Module pointmod

```
MODULE pointmod
...
  INTERFACE init
    MODULE PROCEDURE init_r,init_i
  END INTERFACE

CONTAINS
...
  SUBROUTINE init_r(p,x,y)
    IMPLICIT NONE
    TYPE (point):: p
    REAL (prec):: x,y

    p%x=x
    p%y=y
  END SUBROUTINE init_r
```

```
SUBROUTINE init_i(p,x,y)
  IMPLICIT NONE
  TYPE (point):: p
  integer:: x,y
```

```
  p%x=x
  p%y=y
```

```
END SUBROUTINE init_i
```

...

## In the main program

...

```
  CALL init(p1,1.0_prec,2.0_prec)
```

...

```
  CALL init(p2,3,4)
```

...

2. The module `swap`: Modified from the original `swapmod.f90` by adding specific switch routines and changing the variable name “second” to “next”, because “second” is the name of an intrinsic function of gfortran. The main program is otherwise similar as `swap.f90`, but the character string “second” is changed to “next”.

```
MODULE swap

  IMPLICIT NONE

  INTERFACE switch
    MODULE PROCEDURE switch_i, switch_r, &
      switch_k, switch_a, switch_a2
  END INTERFACE
```

CONTAINS

```
SUBROUTINE switch_i(first, next)
  IMPLICIT NONE
  INTEGER :: first, next, temp
  temp = next
  next = first
  first = temp
END SUBROUTINE switch_i
```

```
SUBROUTINE switch_r(first, next)
  IMPLICIT NONE
  REAL :: first, next, temp
  temp = next
  next = first
  first = temp
END SUBROUTINE switch_r
```

```
SUBROUTINE switch_k(first, next)
  IMPLICIT NONE
  CHARACTER(LEN=10) :: first, next, temp
  temp = next
  next = first
  first = temp
END SUBROUTINE switch_k
```

```
SUBROUTINE switch_a(first, next)
  IMPLICIT NONE
  INTEGER, DIMENSION(:):: first, next
  INTEGER, DIMENSION(SIZE(first)):: temp
  temp = next
  next = first
  first = temp
END SUBROUTINE switch_a
```

```
SUBROUTINE switch_a2(first, next)
  IMPLICIT NONE
  INTEGER, DIMENSION(:,):: first, next
  INTEGER, DIMENSION(SIZE(first,1),SIZE(first,2)):: temp
  temp = next
  next = first
  first = temp
END SUBROUTINE switch_a2
```

```
END MODULE swap
```

# Exercise 14

## 1. The module `resistor`

```
MODULE resistor
  IMPLICIT NONE
  INTERFACE operator(.series.)
    MODULE PROCEDURE inseries
  END INTERFACE
  INTERFACE operator(.parallel.)
    MODULE PROCEDURE paralleling
  END INTERFACE

  CONTAINS
  FUNCTION inseries(r1,r2) RESULT(r)
    IMPLICIT NONE
    REAL, INTENT(in):: r1, r2
    REAL:: r
    r=r1+r2
  END FUNCTION inseries
```

```
FUNCTION paralleling(r1,r2) RESULT@
  IMPLICIT NONE
  REAL, INTENT(in):: r1, r2
  REAL:: r

  r=1/(1/r1+1/r2)

END FUNCTION paralleling

END MODULE resistor
```

## The module `pointmod`

```
MODULE pointmod

...

INTERFACE operator(+)
  MODULE PROCEDURE sumup
END INTERFACE

INTERFACE operator(.distance.)
  MODULE PROCEDURE remote
END INTERFACE

CONTAINS

  FUNCTION sumup(p1,p2) RESULT(p)
    IMPLICIT NONE
    TYPE (point), INTENT(in):: p1, p2
    TYPE (point):: p
    p%x=p1%x+p2%x
    p%y=p1%y+p2%y
  END FUNCTION sumup
```

```
FUNCTION remote(p) RESULT(d)
  IMPLICIT NONE
  TYPE (point), INTENT(in):: p
  REAL (dp):: d

  d=SQRT(p%x**2+p%y**2)
END FUNCTION remote
```

...

```
END MODULE pointmod
```

## In the main program

```
...
TYPE (point):: p, p1, p2
...
p=p1+p2
...
PRINT *, .distance. p
```

# Exercise 15

1. If the compilation fails or results of the program are not correct there is probably a bug or bugs in the program. Try to use also FORTRAN 77 compiler (`g77`).
2. If modification does not succeed, the reason to this is very often the fact that spaces are meaningful in free form. The use of the structures `DO . . .`  
`END DO`, etc. is recommended for readability.
3. FORTRAN 77 does not give error messages, neither Fortran 90, if the file suffix is `.f`, because F90 compilers nowadays recognize then FORTRAN 77 programs. If the free form option is used with the F90 compilers, then it gives syntax error messages, which depends on the used compiler (GCC, PGI, etc.). Add line feeds to a suitable places.

```
PROGRAM test
  IMPLICIT NONE
  LOGICAL :: l = .false.
  REAL :: z = 0.0, y = 1.0
  IF (l) THEN
    z = 1.0
  ELSE
    y = z
  END IF
  PRINT *, y, z
END PROGRAM test
```

# Exercise 16



This is a bit tidied version of the program `fzero`. There are still couple addresses, which one could try to remove by rearranging the code. In fact almost the corresponding FORTRAN 77 program can be found from the book *Numerical Recipes* (Chapter 9.3, subprogram ZBRENT). It is more readable than the Kahaner's version and it is easily modifiable.

```
program test
  implicit none
  external f
  real :: f, a,b,r
  integer :: ifl
  a=0.5
  b=2.0
  r=1.5
  call fzero(f,a,b,r, 1.0e-4, 1.0e-4, ifl)
  WRITE (*,*)a,b,r,ifl
end

real function f(x)
  real x
  f=x**3-1.0
end
```

```
subroutine fzero(f,b,c,r,re,ae,iflag)
! ...
  implicit none
  real, external :: f
  real, intent(inout) :: b,c,r,re,ae
  integer, intent(out) :: iflag
  real :: a,acbs,acmb,aw,cmb,er, &
         fa,fb,fc,fx,fz,p,q, &
         rw,t,tol,z
  integer :: ic,kount

  er = 2.0e0 * epsilon(r)

! Initialize
  z=r
  if(r.le.min(b,c).or.r.ge.max(b,c)) z=c
  rw=max(re,er)
  aw=max(ae,0.0)
  ic=0
  t=z
  fz=f(t)
  fc=fz
  t=b
  fb=f(t)
  kount=2
```

```
if(sign(1.0e0,fz).ne.sign(1.0e0,fb)) then
  c=z
else if(z.ne.c) then
  t=c
  fc=f(t)
  kount=3
  if(sign(1.0e0,fz).ne.sign(1.0e0,fc)) then
    b=z
    fb=fz
  end if
end if
a=c
fa=fc
acbs=abs(b-c)
fx=max(abs(fb),abs(fc))

loop: do
  if (abs(fc) .lt. abs(fb)) then
    a=b
    fa=fb
    b=c
    fb=fc
    c=a
    fc=fa
  end if
```

```
cmb=0.5*(c-b)
acmb=abs(cmb)
tol=rw*abs(b)+aw

! test stopping criterion and function count
if (acmb .le. tol) then
  iflag=1
  if (sign(1.0,fb) .eq. sign(1.0,fc)) then
    iflag = 4
  else if (abs(fb) .gt. fx) then
    iflag = 3
  end if
  exit loop
endif
if(fb.eq.0.e0) then
  iflag=2
  exit loop
end if
if(kount.ge.500) then
  iflag=3
  exit loop
end if
```

```
! calculate new iterate implicitly as b+p/q
! where we arrange p .ge. 0.
! the implicit form is used to prevent overflow.
p=(b-a)*fb
q=fa-fb
if (p .lt. 0.) then
    p=-p
    q=-q
end if

! update a and check for satisfactory reduction
! in the size of the bracketing interval.
! if not, perform bisection.
a=b
fa=fb
ic=ic+1
if (ic .ge. 4) then
    if (8.*acmb .ge. acbs) go to 8
    ic=0
    acbs=acmb
end if
```

```
! test for too small a change
if (p .le. abs(q)*tol) then
  ! increment by tolerance
  b=b+sign(tol,cmb)
  go to 9
else
  ! root ought to be between b and (c+b)/2.
  if (p .ge. cmb*q) go to 8
  ! use secant rule
  b=b+p/q
  go to 9
end if
! use bisection
8 b=0.5*(c+b)
! have completed computation for new iterate b
9 t=b
fb=f(t)
kount=kount+1
! decide whether next step is interpolation or extrapolation
if (sign(1.0,fb) .ne. sign(1.0,fc)) cycle
c=a
fc=fa
end do loop
return
end
```