24

An Introduction to Modules

"Common sense is the best distributed commodity in the world, for every man is convinced that he is well supplied with it."

Descartes

Aims

The aims of this chapter are to look at the facilities found in Fortran provided by modules, in particular:

- The use of a module to aid in the consistent definition of precision throughout a program and subprograms.
- The use of modules for global data.
- The use of modules for derived data types.
- Two examples showing the use of modules with contained proecures and their use to package procedures.
- A complete numerical example solving systems of linear equations using Gaussian elimination.

24 An Introduction to Modules

As summarised in the Chapter 23 we now have the tools to solve many problems using just a main program and one or more external and internal subprograms. Both external and internal subprograms communicate through their argument lists, whilst internal subprograms have access to data in their host program units.

We now introduce another type of program unit, the module, which is probably one of the most important features of Fortran 90. The purpose of modules is quite different from that of subprograms. In their simplest form they exist so that anything required by more than one program unit may be packaged in a module and made available where needed.

The form of a module is

```
MODULE module_name
...
END MODULE module_name
```

and the information contained within it is made available in the program units that need to access it by

```
USE module_name
```

The USE statement must be the first statement after the PROGRAM or SUBROUTINE or FUNCTION statement.

In this chapter we will look at:

- Modules for global data.
- Modules for derived types.
- Modules for explicit interfaces.
- Modules containing procedures.

Modules are another program unit and exist so that anything required by more than one program unit may be packaged in a module and made available where needed.

24.1 Modules for global data

So far the only way that a program unit can communicate with a procedure is through the argument list. Sometimes this is very cumbersome, especially if a number of procedures want access to the same data, and it means long argument lists. The problem can be solved using modules; e.g., by defining the precision to which you wish to work and any constants defined to that precision which may be needed by a number of procedures.

24.2 Modules for precision specification and constant definition

In the following example we use a module to define a parameter Long to specify the precision to which we wish to work, and another for a range of mathematical constants including a value for the parameter π. Note that the parameter π is defined to this working precision. We then import the module defining these parameters into the program units that need them:

```
module precision_definition
  implicit none
  integer , parameter ::
long=selected_real_kind(15,307)
end module precision_definition

module maths_constants
  use precision_definition
  implicit none
  real (long) , parameter :: c = 299792458.0_long
  ! units m s-1
  real (long) , parameter :: &
    e = 2.71828182845904523_long
  real (long) , parameter :: g = 9.812420_long
  ! 9.780 356 m s-2 at sea level on the equator
  ! 9.812 420 m s-2 at sea level in London
  ! 9.832 079 m s-2 at sea level at the poles
  real (long) , parameter :: &
    pi = 3.14159265358979323_long
end module maths_constants

PROGRAM ch2401
  USE Precision_definition
  IMPLICIT NONE
  INTERFACE
    SUBROUTINE Sub1(Radius,Area,Circum)
    USE Precision_definition
    IMPLICIT NONE
    REAL(Long),INTENT(IN)::Radius
    REAL(Long),INTENT(OUT)::Area,Circum
    END SUBROUTINE Sub1
  END INTERFACE
  REAL(Long)::R,A,C
  INTEGER ::I
  DO I=1,10
```

```
    PRINT*,'Radius?'
    READ*,R
    CALL Sub1(R,A,C)
    PRINT *,' For radius    = ',R
    PRINT *,' Area          = ',A
    PRINT *,' Circumference = ',C
  END DO
END PROGRAM ch2401

SUBROUTINE Sub1(Radius,Area,Circum)
  USE Precision_definition
  use maths_constants
  IMPLICIT NONE
  REAL(Long),INTENT(IN)::Radius
  REAL(Long),INTENT(OUT)::Area,Circum
  Area=Pi*Radius*Radius
  Circum=2.0_Long*Pi*Radius
END SUBROUTINE Sub1
```

24.2.1 Note

In this example we wish to work with the precision specified by the kind type parameter Long in the module Precision_definition. In order to do this we use the statement

```
USE precision_definition
```

inside the program unit before any declarations. The kind type parameter Long is then used with all the REAL type declaration e.g.,

```
REAL (Long):: R ,A,C
```

To make sure that all floating point calculations are performed to the working precision specified by Long any constants such as 2.0 in subroutine Sub1 are specified as const_Long e.g.,

```
2.0_Long
```

Note also that we define things once and use them on two occasions, i.e., we define the precision once and use this definition in both the main program and the subroutine.

24.3 Modules for sharing arrays of data

The following example uses one module containing a number of constants and a second module containing an array definition:

```
module data
  implicit none
  integer , parameter    :: n=12
  real , dimension(1:n) :: rainfall
  real , dimension(1:n) :: sorted
end module data

program ch2402
use data
implicit none

  call readdata
  call sortdata
  call printdata

end program ch2402

subroutine readdata
use data
implicit none
integer :: i
character (len=40) :: filename
  print *,' What is the filename ?'
  read *,filename
  open(unit=100,file=filename)
  do i=1,n
    read (100,*) rainfall(i)
  end do
end subroutine readdata

subroutine sortdata
use data
  sorted=rainfall
  call selection

  contains

    subroutine selection
```

```
     implicit none
     integer :: i,j,k
     real :: minimum
       do i=1,n-1
         k=i
         minimum=sorted(i)
         do j=i+1,n
           if (sorted(j) < minimum) then
             k=j
             minimum=sorted(k)
           end if
         end do
         sorted(k)=sorted(i)
         sorted(i)=minimum
       end do
     end subroutine selection

end subroutine sortdata

subroutine printdata
use data
implicit none
integer :: i
  print *,' original data is '
  do i=1,n
    print 100,rainfall(i)
    100 format(1x,f7.1)
  end do
  print *,' Sorted data is '
  do i=1,n
    print 100,sorted(i)
  end do
end subroutine printdata
```

Note that in this example the calls to the subroutines have no parameters. They work with the data contained in the module.

24.4 Modules for derived data types

When using derived data types and passing them as arguments to subroutines, both the actual arguments and dummy arguments must be of the same type, i.e., they must be declared with reference to the same type definition. The only way this can be achieved is by using modules. The user defined type is declared in a module and each program unit that requires that type **uses** the module.

24.4.1 Person data type

In this example we have a user defined type Person which we wish to use in the main program and pass arguments of this type to the subroutines Read_data and Stats. In order to have the type Person available to two subroutines and the main program we have defined Person in a module Personal_details and then made the module available to each program unit with the statement

```
USE Personal_details
```

We also have the use of an interface block to provide the ability to develop the overall solution in stages:

```
MODULE Personal_details
   IMPLICIT NONE
   TYPE Person
      REAL:: Weight
      INTEGER :: Age
      CHARACTER :: Sex
   END TYPE Person
END MODULE Personal_details

PROGRAM ch2403
   USE Personal_details
   IMPLICIT NONE
   INTEGER ,PARAMETER:: Max_no=100
   TYPE (Person), DIMENSION(1:Max_no) :: Patient
   INTEGER :: No_of_patients
   REAL :: Male_average, Female_average
INTERFACE

   SUBROUTINE Read_data(Data,Max_no,No)
      USE Personal_details
      IMPLICIT NONE
      TYPE (Person), DIMENSION (:), INTENT(OUT):: Data
      INTEGER, INTENT(OUT):: No
      INTEGER, INTENT(IN):: Max_no
   END SUBROUTINE Read_Data

   SUBROUTINE Stats(Data,No,M_a,F_a)
      USE Personal_details
```

```
      IMPLICIT NONE
      TYPE(Person), DIMENSION (:) :: Data
      REAL:: M_a,F_a
      INTEGER :: No
   END SUBROUTINE Stats

END INTERFACE
!
   CALL Read_data(Patient,Max_no,No_of_patients)
   CALL Stats( Patient , No_of_patients , &
               Male_average , Female_average)
   PRINT*, 'Average male weight is ',Male_average
   PRINT*, 'Average female weight is ',Female_average
END PROGRAM ch2403

SUBROUTINE Read_Data(Data,Max_no,No)
   USE Personal_details
   IMPLICIT NONE
   TYPE (PERSON), DIMENSION (:), INTENT(OUT)::Data
   INTEGER, INTENT(OUT):: No
   INTEGER, INTENT(IN):: Max_no
   INTEGER :: I
   DO
      PRINT *,'Input number of patients'
      READ *,No
      IF ( No > 0 .AND. No <= Max_no) EXIT
   END DO
   DO I=1,No
      PRINT *,'For person ',I
      PRINT *,'Weight ?'
      READ*,Data(I)%Weight
      PRINT*,'Age ?'
      READ*,Data(I)%Age
      PRINT*,'Sex ?'
      READ*,Data(I)%Sex
   END DO
END SUBROUTINE Read_Data

SUBROUTINE Stats(Data,No,M_a,F_a)
   USE Personal_details
   IMPLICIT NONE
   TYPE(Person), DIMENSION(:)::Data
```

```
   REAL :: M_a,F_a
   INTEGER:: No
   INTEGER :: I,No_f,No_m
   M_a=0.0; F_a=0.0;No_f=0; No_m =0
   DO I=1,No
      IF ( Data(I)%Sex == 'M' &
      .OR. Data(I)%Sex == 'm') THEN
         M_a=M_a+Data(I)%Weight
         No_m=No_m+1
      ELSEIF(Data(I)%Sex == 'F' &
         .OR. Data(I)%Sex == 'f') THEN
         F_a=F_a +Data(I)%Weight
         No_f=No_f+1
      ENDIF
   END DO
   IF (No_m > 0 ) THEN
      M_a = M_a/No_m
   ENDIF
   IF (No_f > 0 ) THEN
      F_a = F_a/No_f
   ENDIF
END SUBROUTINE Stats
```

24.5 Modules containing procedures — Quicksort example

In this example we rewrite the Quicksort example to use modules. Each subroutine is put into a module on its own. The program is given below:

```
module read_data

contains

  SUBROUTINE Read(File_Name,Raw_Data,How_Many)
  IMPLICIT NONE
  CHARACTER (LEN=*) , INTENT(IN) :: File_Name
  INTEGER , INTENT(IN) :: How_Many
  REAL , INTENT(OUT) , DIMENSION(:) :: Raw_Data

  INTEGER :: I

    OPEN(FILE=File_Name,UNIT=1)
    DO I=1,How_Many
      READ (UNIT=1,FMT=*) Raw_Data(I)
```

```
    ENDDO
  END SUBROUTINE Read

end module read_data

module sort_data

contains

  SUBROUTINE Sort(Raw_Data,How_Many)
  IMPLICIT NONE
  INTEGER , INTENT(IN) :: How_Many
  REAL , INTENT(INOUT) , DIMENSION(:) :: Raw_data

    CALL QuickSort(1,How_Many)

  CONTAINS

    RECURSIVE SUBROUTINE QuickSort(L,R)
    IMPLICIT NONE
    INTEGER , INTENT(IN) :: L,R
    INTEGER :: I,J
    REAL :: V,T

    i=l
    j=r
    v=raw_data( int((l+r)/2) )
    do
      do while (raw_data(i) < v )
          i=i+1
      enddo
      do while (v < raw_data(j) )
          j=j-1
      enddo
      if (i<=j) then
        t=raw_data(i)
        raw_data(i)=raw_data(j)
        raw_data(j)=t
        i=i+1
        j=j-1
      endif
      if (i>j) exit
```

```
    enddo

    if (l<j) then
      call quicksort(l,j)
    endif

    if (i<r) then
      call quicksort(i,r)
    endif

    END SUBROUTINE QuickSort

  END SUBROUTINE Sort

end module sort_data

module print_data

contains

  SUBROUTINE Print(Raw_Data,How_Many)
  IMPLICIT NONE
  INTEGER , INTENT(IN) :: How_Many
  REAL , INTENT(IN) , DIMENSION(:) :: Raw_data
  INTEGER :: I
    OPEN(FILE='SORTED.DAT',UNIT=2)
    DO I=1,How_Many
      WRITE(UNIT=2,FMT=*) Raw_data(I)
    ENDDO
    CLOSE(2)
  END SUBROUTINE Print

end module print_data

PROGRAM ch2404
use read_data
use sort_data
use print_data
IMPLICIT NONE
INTEGER                                     :: How_Many
CHARACTER   (LEN=20)                       :: File_Name
REAL , ALLOCATABLE , DIMENSION(:)   :: Raw_data
```

```
integer , dimension(8)                :: dt

  PRINT * , ' How many data items are there?'
  READ  * , How_Many
  PRINT * , ' What is the file name?'
  READ '(A)',File_Name

  call date_and_time(values=dt)
  PRINT 100 , dt(6),dt(7),dt(8)
  100 FORMAT(' Initial cpu time     = ',3(2x,i10))

  ALLOCATE(Raw_data(How_Many))

  call date_and_time(values=dt)
  PRINT 110 , dt(6),dt(7),dt(8)
  110 FORMAT(' Allocate cpu time    = ',3(2x,i10))

  CALL Read(File_Name,Raw_Data,How_Many)

  call date_and_time(values=dt)
  PRINT 120 , dt(6),dt(7),dt(8)
  120 FORMAT(' Read data cpu time   = ',3(2x,i10))

  CALL Sort(Raw_Data,How_Many)

  call date_and_time(values=dt)
  PRINT 130 , dt(6),dt(7),dt(8)
  130 FORMAT(' Quick sort cpu time = ',3(2x,i10))

  CALL Print(Raw_Data,How_Many)

  call date_and_time(values=dt)
  PRINT 140 , dt(6),dt(7),dt(8)
  140 FORMAT(' Write data cpu time = ',3(2x,i10))

  PRINT * , ' '
  PRINT *, ' Data written to file SORTED.DAT'

END PROGRAM ch2404
```

The keys in this example is that each subroutine is in a module as a contained procedure and we just have three use statements in the main program to make the subroutines available.

Note that we do not now have any interface blocks in this program. The cross unit checking that interface blocks make available is provided automatically when using modules.

24.6 Modules containing procedures — Statistics example

This is a reworking of the statistics subroutine introduced earlier. We now break the subroutine down into three separate functions:

- mean
- std_dev
- median

that are contained within a module. The median function also has its own internal procedure, find.

```
module statistics

contains

   real function mean(x,n)
   implicit none
   integer , intent(in)                        :: n
   real     , intent(in)  , dimension(:)       :: x
   integer :: i
   real     :: total
      total=0
      do i=1,n
         total=total+x(i)
      end do
      mean=total/n
   end function mean

   real function std_dev(x,n,mean)
   integer , intent(in)                        :: n
   real     , intent(in)  , dimension(:)       :: x
   real     , intent(in)                       :: mean
   real                                        :: variance
      variance=0
      do i=1,n
```

```
      variance=variance + (x(i)-mean)**2
    end do
    variance=variance/(n-1)
    std_dev=sqrt(variance)
  end function std_dev

  real function median(x,n)
  integer , intent(in)                        :: n
  real     , intent(in) , dimension(:)        :: x
  real     , dimension(1:n)                   :: y
    y=x
    if (mod(n,2) == 0) then
      median = ( find(n/2)+find((n/2)+1) )/2
    else
      median=find((n/2)+1)
    endif

  contains

    real function find(k)
    implicit none
    integer , intent(in) :: k
    integer :: l,r,i,j
    real :: t1,t2
      l=1
      r=n
      do while (l<r)
        t1=y(k)
        i=l
        j=r
        do
          do while (y(i)<t1)
            i=i+1
          end do
          do while (t1<y(j))
            j=j-1
          end do
          if (i<=j) then
            t2=y(i)
            y(i)=y(j)
            y(j)=t2
            i=i+1
```

```
          j=j-1
        end if
        if (i>j) exit
      end do
      if (j<k) then
        l=i
      end if
      if (k<i) then
        r=j
      end if
    end do
    find=y(k)
  end function find

  end function median

end module statistics

program ch2405

use statistics

implicit none
integer :: n
real , allocatable , dimension(:) :: x
real :: m,sd,med
integer , dimension(8) :: v

  print *,' How many values ?'
  read *,n
  call date_and_time(values=v)
  print *,' initial                    ',v(6),v(7),v(8)
  allocate(x(1:n))
  call date_and_time(values=v)
  print *,' allocate                   ',v(6),v(7),v(8)
  call random_number(x)
  call date_and_time(values=v)
  print *,' random                     ',v(6),v(7),v(8)
  x=x*1000
  call date_and_time(values=v)
  print *,' output                     ',v(6),v(7),v(8)
  m=mean(x,n)
```

```
   call date_and_time(values=v)
   print *,' mean                          ',v(6),v(7),v(8)
   print *,' mean                       = ',m
   sd=std_dev(x,n,m)
   call date_and_time(values=v)
   print *,' standard deviation    ',v(6),v(7),v(8)
   print *,' Standard deviation = ',sd
   med = median(x,n)
   call date_and_time(values=v)
   print *,' median                        ',v(6),v(7),v(8)
   print *,' median is               = ',med
end program ch2405
```

Note again that we do not need to have explicit interface blocks as the packaging of the procedures within a module provides the interface checking automatically.

The program also has timing code added to allow profiling of the various parts of the program.

24.7 The solution of linear equations using Gaussian elimination

At this stage we have introduced many of the concepts needed to write numerical code, and have included a popular algorithm, Gaussian elimination, together with a main program which uses it and a module to bring together many of the features covered so far.

Finding the solution of a system of linear equations is very common in scientific and engineering problems, either as a direct physical problem or indirectly, for example, as the result of using finite difference methods to solve a partial differential equation. We will restrict ourselves to the case where the number of equations and the number of unknowns are the same. The problem can be defined as:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{12}x_2 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
\ldots \quad \ldots \quad \ldots \quad \ldots \quad &= \ldots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}
\tag{1}
$$

or

$$
\begin{pmatrix}
a_{11} & a_{12} & \ldots & a_{1n} \\
a_{21} & a_{22} & \ldots & a_{2n} \\
\ldots & \ldots & \ldots & \ldots \\
a_{n1} & a_{n2} & \ldots & a_{nn}
\end{pmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix}
=
\begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{pmatrix}
$$

which can be written as:

$$A\,x = b \tag{2}$$

where A is the $n \times n$ coefficient matrix, b is the right-hand-side vector and x is the vector of unknowns. We will also restrict ourselves to the case where A is a general real matrix.

Note that there is a unique solution to (2) if the inverse, A^{-1}, of the coefficient matrix A, exists. However, the system should never be solved by finding A^{-1} and then solving $A^{-1}\ b = x$ because of the problems of rounding error and the computational costs.

A well-known method for solving (2) is Gaussian elimination, where multiples of equations are subtracted from others so that the coefficients below the diagonal become zero, producing a system of the form:

$$\begin{pmatrix} a_{11}^* & a_{12}^* & \cdots & a_{1n}^* \\ 0 & a_{22}^* & \cdots & a_{2n}^* \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & a_{nn}^* \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^* \\ b_2^* \\ \cdots \\ b_n^* \end{pmatrix}$$

where A has been transformed into an upper triangular matrix. By a process of *backward substitution* the values of *x drop* out.

The subroutine Gaussian_Elimination implements the Gaussian elimination algorithm with *partial pivoting,* which ensure that the multipliers are less than 1 in magnitude, by interchanging rows if necessary. This is to try and prevent the buildup of errors.

This implementation is based on two LINPACK routines SGEFA and SGESL and a Fortran 77 subroutine written by Tim Hopkins and Chris Phillips and found in their book *Numerical Methods in Practice.*

The matrix A and vector B are passed to the subroutine Gaussian_Elimination and on exit both A and B are overwritten. Mathematically Gaussian elimination is described as working on rows, and using partial pivoting row interchanges may be necessary. Due to Fortran's row element ordering, to implement this algorithm efficiently it works on columns rather than rows by interchanging elements within a column if necessary.

```
MODULE Precisions
INTEGER,PARAMETER:: Long=SELECTED_REAL_KIND(15,307)
END MODULE Precisions

PROGRAM Solve
```

```
   USE Precisions
   IMPLICIT NONE
   INTEGER :: I,N
   REAL (Long), ALLOCATABLE:: A(:,:),B(:),X(:)
   LOGICAL:: Singular

INTERFACE

   SUBROUTINE Gaussian_Elimination(A,N,B,X,Singular)
      USE Precisions
      IMPLICIT NONE
      INTEGER, INTENT(IN)::N
      REAL (Long), INTENT (INOUT) :: A(:,:),B(:)
      REAL (Long), INTENT(OUT)::X(:)
      LOGICAL, INTENT(OUT) :: Singular
   END SUBROUTINE Gaussian_Elimination

END INTERFACE

   PRINT *,'Number of equations?'
   READ *,N
   ALLOCATE(A(1:N,1:N),B(1:N),X(1:N))
   DO I=1,N
      PRINT *,'Input elements of row ',I,' of A'
      READ*,A(I,1:N)
      PRINT*,'Input element ',I,' of B'
      READ *,B(I)
   END DO
   CALL Gaussian_Elimination(A,N,B,X,Singular)
   IF(Singular) THEN
      PRINT*, 'Matrix is singular'
   ELSE
      PRINT*, 'Solution X:'
      PRINT*,X(1:N)
   ENDIF
END PROGRAM Solve

SUBROUTINE Gaussian_Elimination(A,N,B,X,Singular)
! Routine to solve a system Ax=b
! using Gaussian Elimination
! with partial pivoting
```

```
! The code is based on the Linpack routines
! SGEFA and SGESL
! and operates on columns rather than rows!
   USE Precisions
   IMPLICIT NONE
! Matrix A and vector B are over-written
! Arguments
   INTEGER, INTENT(IN):: N
   REAL (Long),INTENT(INOUT):: A(:,:),B(:)
   REAL (Long),INTENT(OUT)::X(:)
   LOGICAL,INTENT(OUT)::Singular
! Local variables
   INTEGER::I,J,K,Pivot_row
   REAL (Long):: Pivot,Multiplier,Sum,Element
   REAL (Long),PARAMETER::Eps=1.E-13_Long
!
! Work through the matrix column by column
!
   DO K=1,N-1
!
!  Find largest element in column K for pivot
!
   Pivot_row = MAXVAL( MAXLOC( ABS( A(K:N,K) ) ) ) &
      + K - 1
!
! Test to see if A is singular
! if so return to main program
!
      IF(ABS(A(Pivot_row,K)) <= Eps) THEN
         Singular=.TRUE.
         RETURN
      ELSE
         Singular = .FALSE.
      ENDIF
!
! Exchange elements in column K if largest is
! not on the diagonal
!
      IF(Pivot_row /= K) THEN
         Element=A(Pivot_row,K)
         A(Pivot_Row,K)=A(K,K)
         A(K,K)=Element
```

```
         Element=B(Pivot_row)
         B(Pivot_row)=B(K)
         B(K)=Element
      ENDIF
!
! Compute multipliers
! elements of column K below diagonal
! are set to these multipliers for use
! in elimination later on
!
      A(K+1:N,K) = A(K+1:N,K)/A(K,K)
!
! Row elimination performed by columns for efficiency
!
      DO J=K+1,N
         Pivot = A(Pivot_row,J)
         IF(Pivot_row /= K) THEN
!          Swap if pivot row is not K
            A(Pivot_row,J)=A(K,J)
            A(K,J)=Pivot
         ENDIF
         A(K+1:N,J)=A(K+1:N,J)-Pivot* A(K+1:N,K)
      END DO
!
! Apply same operations to B
!
      B(K+1:N)=B(K+1:N)-A(K+1:N,K)*B(K)
   END DO
!
! Backward substitution
!
   DO I=N,1,-1
      Sum = 0.0
      DO J= I+1,N
         Sum=Sum+A(I,J)*X(J)
      END DO
      X(I)=(B(I)-Sum)/A(I,I)
   END DO
END SUBROUTINE Gaussian_Elimination
```

24.7.1 Notes

24.7.1.1 Module for kind type

A module, Precisions, has been used to define a kind type parameter, Long, to specify the floating point precision to which we wish to work. This module is then used by the main program and the subroutine, and the kind type parameter Long is used with all the REAL type definitions and with any constants, e.g.,

```
REAL(Long), PARAMETER :: Eps=1.E-13_Long
```

24.7.1.2 Deferred-shape arrays

In the main program matrix A and vectors B and X are declared as deferred-shape arrays, by specifying their rank only and using the ALLOCATABLE attribute. Their shape is determined at run time when the variable N is read in and then the statement

```
ALLOCATE(A(1:N,1:N), B(1:N), X(1:N))
```

is used.

24.7.1.3 Intrinisic functions MAXVAL and MAXLOC

In the context of subroutine Gaussian_Elimination we have used:

```
MAXVAL ( MAXLOC (ABS ( A ( K:N,K ) ) ) ) + K - 1
```

Breaking this down,

```
MAXLOC ( ABS ( A (K:N,K) ) )
```

takes the rank 1 array

$$(|A(K,K)|,|A(K+1,K)|,\ldots|A(N,K)|) \qquad (1)$$

where $|A(K,K)| =$ ABS(*A(K,K)*) and of length *N- K + 1*. It returns the position of the largest element as a rank 1 array of size one, e.g., (*L*)

Applying MAXVAL to this rank 1 array (*L*) returns *L* as a scalar, *L* being the position of the largest element of array (1).

What we actually want is the position of the largest element of (1) , but in the K^{th} column of matrix *A*. We therefore have to add *K-1* to *L* to give the actual position in column *K* of A.

24.8 Notes on module usage and compilation

If we only have one file comprising all of the program units (main program, modules, functions and subroutines) then there is little to worry about. However, it is recommended that larger-scale programs be developed as a collection of files with related program units in each file, or even one program unit per file. This is more productive in the longer term, but it will lead to problems with modules unless we compile each module **before** we use it in other program units.

Secondly, we must use one directory or subdirectory so that the compiler and linker can find each program unit.

Thirdly, we must be aware of the file naming conventions used by each compiler implementation we work with. Consider the following:

Fortran module name	Compaq under DOS	NAG f95 Sun Ultra Sparc
Precisions	Precisions.mod	Precisions.mod

Whilst in this case they are the same, this is not guaranteed.

24.9 Summary

We have now introduced the concept of a module, another type of program unit, probably one of of the most important features of Fortran 90. We have seen in this chapter how they can be used:

- Define global data.
- Define derived data types.
- Contain explicit procedure interfaces.
- Cackage together procedures.

This is a very powerful addition to the language, especially when constructing large programs and procedure libraries.

24.10 Problems

1. Write two functions, one to calculate the volume of a cylinder $\pi r^2 l$ where the radius is r and the length is l, and the other to calculate the area of the base of the cylinder πr^2. Define π as a parameter in a module which is used by the two functions. Now write a main program which prompts the user for the values of r and l, calls the two functions and prints out the results.

2. Make all the real variables in the above problem have 15 significant digits and a range of 10^{-307} to 10^{+307}. Use a module.

24.11 Bibliography

Dongarra, J., Bunch, J.R., Moler, C.B., and Stewart, G.W. *LINPACK User's Guide*. SIAM Publications, 1979.

- This Fortran 77 package is for the solution of simultaneous systems of linear algebraic equations. Special subroutines are included for many common types of coefficient matrices. The source is available through NETLIB. See Chapter 28 for more details.

Hopkins T., Phillips C., *Numerical Methods in Practice, using the NAG Library*. Addison-Wesley.

- This is a very good practical introduction to numerical analysis, with the aim of guiding users to the more commonly used routines in the NAG Fortran 77 library. It does this by introducing topics, giving some background, advantages and disadvantages, and the Fortran 77 code for some of the more well-known algorithms. It then introduces the appropriate NAG routine with a brief discussion of its use, calling sequence and any error reporting facilities. We've found this invaluable for many of our students who are users of the NAG library but not well versed with numerical analysis.
  Maybe we will see a Fortran 90 version of this book in the near future?

NAG. Visit their web site for up to date details of their products:

- http://www.nag.co.uk/

Visual Numerics. Visit their web site for details of their products:

- http://www.vni.com/index.html