22 Introduction to Subroutines

"A man should keep his brain attic stacked with all the furniture he is likely to use, and the rest he can put away in the lumber room of his library, where he can get at it if he wants."

Sir Arthur Conan Doyle, *Five Orange Pips*

Aims

The aims of this chapter are:

- To consider some of the reasons for the inclusion of subroutines in a programming language.
- To introduce with a concrete example some of the concepts and ideas involved with the definition and use of subroutines.
    - The INTERFACE statement and interface blocks.
    - Arguments or parameters.
    - The INTENT attribute for parameters.
    - The CALL statement.
    - Scope of variables.
    - Local variables and the SAVE attribute.
    - The use of parameters to report on the status of the action carried out in the subroutine.

22 Introduction to Subroutines

In the earlier chapter on functions we introduced two types of function

- Intrinsic functions — which are part of the language.
- User defined functions — by which we extend the language.

We now introduce subroutines which collectively with functions are given the name procedures. Procedures provide a very powerful extension to the language by:

- Providing us with the ability to break problems down into simpler more easily solvable subproblems.
- Allowing us to concentrate on one aspect of a problem at a time.
- Avoiding duplication of code.
- Hiding away messy code so that a main program is a sequence of calls to procedures.
- Providing us with the ability to put together collections of procedures that solve commonly occurring subproblems, often given the name libraries, and generally compiled.
- Allowing us to call procedures from libraries written, tested and documented by experts in a particular field. There is no point in reinventing the wheel!

There are a number of concepts required for the successful use of subroutines and we met some of them in Chapter 15 when we looked at user defined functions. We will extend the ideas introduced there of parameters and introduce the additional concept of an interface block. The ideas are best explained with a concrete example.

Note that we use the terms parameters and arguments interchangeably.

22.1 Example 1

This example is one we met earlier that solves a quadratic equation, i.e., solves $a x^2 + b x + c = 0$

The program to do this originally was just one program. In the example below we break that problem down into smaller parts and make each part a subroutine. The components are:

- Main program or driving routine.
- Interaction with user to get the coefficients of the equation.

- Solution of the quadratic.

Let us look now at how we do this with the use of subroutines:

```
PROGRAM ch2201
IMPLICIT NONE
! Simple example of the use of a main program and two
! subroutines. One interacts with the user and the
! second solves a quadratic equation,
! based on the user input.

REAL :: P, Q, R, Root1, Root2
INTEGER :: IFail=0
LOGICAL :: OK=.TRUE.
   CALL Interact(P,Q,R,OK)
   IF (OK) THEN
      CALL Solve(P,Q,R,Root1,Root2,IFail)
      IF (IFail == 1) THEN
         PRINT *,' Complex roots,
         PRINT *,' calculation abandoned'
      ELSE
         PRINT *,' Roots are  ',Root1,'   ',Root2
      ENDIF
   ELSE
      PRINT*,' Error in data input program ends'
   ENDIF
END PROGRAM ch2201

SUBROUTINE Interact(A,B,C,OK)
   IMPLICIT NONE
   REAL , INTENT(OUT) :: A
   REAL , INTENT(OUT) :: B
   REAL , INTENT(OUT) :: C
   LOGICAL , INTENT(OUT) :: OK
   INTEGER :: IO_Status=0
   PRINT*,' Type in the coefficients A, B AND C'
   READ(UNIT=*,FMT=*,IOSTAT=IO_Status)A,B,C
   IF (IO_Status == 0) THEN
      OK=.TRUE.
   ELSE
      OK=.FALSE.
   ENDIF
END SUBROUTINE Interact
```

```
SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
   IMPLICIT NONE
   REAL , INTENT(IN) :: E
   REAL , INTENT(IN) :: F
   REAL , INTENT(IN) :: G
   REAL , INTENT(OUT) :: Root1
   REAL , INTENT(OUT) :: Root2
   INTEGER , INTENT(INOUT) :: IFail
! Local variables
   REAL :: Term
   REAL :: A2
   Term = F*F - 4.*E*G
   A2 = E*2.0
! if term < 0, roots are complex
   IF(Term < 0.0)THEN
      IFail=1
   ELSE
      Term = SQRT(Term)
      Root1 = (-F+Term)/A2
      Root2 = (-F-Term)/A2
   ENDIF
END SUBROUTINE Solve
```

22.1.1 Defining a subroutine

A subroutine is defined as

```
SUBROUTINE subroutine_name(optional list of dummy arguments)
        IMPLICIT NONE
        dummy argument type definitions with INTENT
        ...
 END SUBROUTINE subroutine_name
```

and from the earlier example we have the subroutine

```
SUBROUTINE Interact(A,B,C,OK)
   IMPLICIT NONE
   REAL, INTENT(OUT)::A,B,C
   LOGICAL, INTENT(OUT)::OK


END SUBROUTINE Interact
```

22.1.2 Referencing a subroutine

To reference a subroutine you use the CALL statement:

CALL subroutine_name(optional list of actual arguments)

and from the earlier example the call to subroutine `Interact` was of the form:

```
CALL Interact(P,Q,R,OK)
```

When a subroutine returns to the calling program unit control is passed to the statement following the CALL statement.

22.1.3 Dummy arguments or parameters and actual arguments

Procedures and their calling program units communicate through their arguments. We often use the terms parameter and arguments interchangeably throughout this text. The SUBROUTINE statement normally contains a list of dummy arguments, separated by commas and enclosed in brackets. The dummy arguments have a type associated with them; for example, in subroutine Solve X is of type REAL, but no space is put aside for this in memory. When the subroutine is referenced e.g., CALL Solve(P,Q,R,Root1,Root2,Ifail), then the dummy argument *points* to the actual argument P, which is a variable in the calling program unit. The dummy argument and the actual argument must be of the same type — in this case REAL.

22.1.4 Intent

It is recommended that dummy arguments have an INTENT attribute. In the earlier example subroutine `Solve` has a dummy argument E with INTENT(IN), which means that when the subroutine is referenced or called it is expecting E to have a value, but its value cannot be changed inside the subroutine. This acts as an extra security measure besides making the program easier to understand. For each parameter it may have one of three attributes:

- INTENT(IN), where the parameter already has a value and cannot be altered in the called routine.
- INTENT(OUT), where the parameter does not have a value, and is given one in the called routine.
- INTENT(INOUT), where the parameter already has a value and this is changed in the called routine.

22.1.5 Local variables

We saw with functions that variables could be essentially local to the function and unavailable elsewhere. The concept of local variables also applies to subroutines. In the example above Term and A2 are both local variables to the subroutine `Solve`.

22.1.6 Local variables and the SAVE attribute

Local variables are usually created when a procedure is called and their value lost when execution returns to the calling program unit. To make sure that a local variable retains its values between calls to a subprogram the SAVE attribute can be used on a type statement; e.g.,

```
INTEGER , SAVE :: I
```

means that when this statement appears in a subprogram the value of the local variable I is saved between calls.

22.1.7 Scope of variables

In most cases variables are only available within the program unit that defines them. The introduction of argument lists to functions and subroutines immediately opens up the possibility of data within one program unit becoming available in one or more other program units.

In the main program we declare the variables P, Q, R, Root1, Root2, IFail and OK.

Subroutine `Interact` has no variables locally declared. It works on the arguments A, B, C and OK; which map onto P, Q, R and OK from the main program, i.e., it works with those variables.

Subroutine `Solve` has two locally defined variables, Term and A2. It works with the variables E, F, G, Root1, Root2 and IFail, which map onto P, Q, R, Root1, Root2 and IFail from the main program.

22.1.8 Status of the action carried out in the subroutine

It is also useful to use parameters that carry information regarding the status of the action carried out by the subroutine. With the subroutine `Interact` we use a logical variable OK to report on the status of the interaction with the user. In the subroutine `Solve` we use the status of the integer variable Ifail to report on the status of the solution of the equation.

22.2 Example 2

Consider the following example:

```
program ch2202
implicit none
real :: a,b,c
  a = 1000.0
  b =    20.0
  call divide(a,b,c)
  print *,c
```

```
end program ch2202

subroutine divide(a,b,c)
implicit none
integer , intent(in) :: a
integer , intent(in) :: b
integer , intent(out):: c
  c=a/b
end subroutine divide
```

There is a fundamental problem here. In the main program the variables A, B and C are declared to be of type real. In the subroutine DIVIDE they are integer.

If the main program and subroutine are in one file when compiled then the compiler has the oportunity of catching this mismatch. The Nag f95 compiler release 4.2 and the Salford FTN95 compiler release 4.6.0 both diagnose this error and will not compile the program. The following compilers compiled and executed the code generating the following answers:

CVF 6.6C:	1.4012985E-45
Intel 9.0	1.4012985E-45
Lahey 5.7	1.40129846E-45

Fortran 90 introduced a number of language features to help in this area:

- Interface blocks.
- Contained procedures.
- Modules.

We will look at the first two in this chapter and at modules later on.

22.3 Example 3 — Quadratic example with interface blocks

This is the first example with the addition of interface blocks:

```
PROGRAM ch2203
IMPLICIT NONE
! Simple example of the use of a main program and two
! subroutines. One interacts with the user and the
! second solves a quadratic equation,
! based on the user input.
INTERFACE

  SUBROUTINE Interact(A,B,C,OK)
```

```
      IMPLICIT NONE
      REAL , INTENT(OUT) :: A
      REAL , INTENT(OUT) :: B
      REAL , INTENT(OUT) :: C
      LOGICAL , INTENT(OUT) :: OK
    END SUBROUTINE Interact

    SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
      IMPLICIT NONE
      REAL , INTENT(IN) :: E
      REAL , INTENT(IN) :: F
      REAL , INTENT(IN) :: G
      REAL , INTENT(OUT) :: Root1
      REAL , INTENT(OUT) :: Root2
      INTEGER , INTENT(INOUT) :: IFail
    END SUBROUTINE Solve

  END INTERFACE

  REAL :: P, Q, R, Root1, Root2
  INTEGER :: IFail=0
  LOGICAL :: OK=.TRUE.
    CALL Interact(P,Q,R,OK)
    IF (OK) THEN
      CALL Solve(P,Q,R,Root1,Root2,IFail)
      IF (IFail == 1) THEN
        PRINT *,' Complex roots, calculation abandoned'
      ELSE
        PRINT *,' Roots are  ',Root1,'  ',Root2
      ENDIF
    ELSE
      PRINT*,' Error in data input program ends'
    ENDIF
  END PROGRAM ch2203

  SUBROUTINE Interact(A,B,C,OK)
    IMPLICIT NONE
    REAL , INTENT(OUT) :: A
    REAL , INTENT(OUT) :: B
    REAL , INTENT(OUT) :: C
    LOGICAL , INTENT(OUT) :: OK
    INTEGER :: IO_Status=0
    PRINT*,' Type in the coefficients A, B AND C'
    READ(UNIT=*,FMT=*,IOSTAT=IO_Status)A,B,C
    IF (IO_Status == 0) THEN
      OK=.TRUE.
    ELSE
      OK=.FALSE.
    ENDIF
  END SUBROUTINE Interact

  SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
    IMPLICIT NONE
    REAL , INTENT(IN) :: E
    REAL , INTENT(IN) :: F
    REAL , INTENT(IN) :: G
    REAL , INTENT(OUT) :: Root1
    REAL , INTENT(OUT) :: Root2
    INTEGER , INTENT(INOUT) :: IFail
  ! Local variables
    REAL :: Term
    REAL :: A2
    Term = F*F - 4.*E*G
    A2 = E*2.0
  ! if term < 0, roots are complex
    IF(Term < 0.0)THEN
      IFail=1
    ELSE
      Term = SQRT(Term)
      Root1 = (-F+Term)/A2
      Root2 = (-F-Term)/A2
    ENDIF
  END SUBROUTINE Solve
```

The key code is given below:

```
INTERFACE

   SUBROUTINE Interact(A,B,C,OK)
      IMPLICIT NONE
      REAL , INTENT(OUT) :: A
      REAL , INTENT(OUT) :: B
      REAL , INTENT(OUT) :: C
      LOGICAL , INTENT(OUT) :: OK
   END SUBROUTINE Interact
   SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
      IMPLICIT NONE
      REAL , INTENT(IN) :: E
      REAL , INTENT(IN) :: F
      REAL , INTENT(IN) :: G
      REAL , INTENT(OUT) :: Root1
      REAL , INTENT(OUT) :: Root2
      INTEGER , INTENT(INOUT) :: IFail
   END SUBROUTINE Solve

END INTERFACE
```

Interface blocks in the above example provide us with the ability to do type checking between the calling routine and the called routine. One of the most common errors in programming is getting the sequence and type of the parameters wrong between subprograms. There is of course the editing overhead of duplicating the code in this example. We will look at software tools that can generate interface blocks for us in a later chapter.

There are times when the use of interface blocks is mandatory in Fortran and we will cover this as and when required. However, it is good working practice to provide interface blocks when dealing with legacy Fortran 77 style code.

We will look at additional ways of providing explicit interfaces later on.

As Fortran 95 libraries become more widely available interface blocks for library routines will be provided by the supplier on line, and this minimises much of the effort in using them. Nag, for example, already has interface blocks available for its library for many platforms.

22.4 Example 4 — Quadratic example and the CONTAINS statement

This example solves the problem of diagnosing mismatches between the calling and called routine by the CONTAINS statement. The two subroutines Interact and Solve are now part of the main program. This method has drawbacks with larger codes suites as we will end up recompiling all of the code within the main program:

```
PROGRAM ch2204
IMPLICIT NONE
! Simple example of the use of a main program and two
! subroutines. One interacts with the user and the
! second solves a quadratic equation,
! based on the use input.

REAL :: P, Q, R, Root1, Root2
INTEGER :: IFail=0
LOGICAL :: OK=.TRUE.
  CALL Interact(P,Q,R,OK)
  IF (OK) THEN
     CALL Solve(P,Q,R,Root1,Root2,IFail)
     IF (IFail == 1) THEN
       PRINT *,' Complex roots, calculation abandoned'
     ELSE
       PRINT *,' Roots are  ',Root1,'  ',Root2
     ENDIF
  ELSE
     PRINT*,' Error in data input program ends'
  ENDIF

contains

SUBROUTINE Interact(A,B,C,OK)
  IMPLICIT NONE
  REAL , INTENT(OUT) :: A
  REAL , INTENT(OUT) :: B
  REAL , INTENT(OUT) :: C
  LOGICAL , INTENT(OUT) :: OK
  INTEGER :: IO_Status=0
  PRINT*,' Type in the coefficients A, B AND C'
  READ(UNIT=*,FMT=*,IOSTAT=IO_Status)A,B,C
  IF (IO_Status == 0) THEN
     OK=.TRUE.
  ELSE
     OK=.FALSE.
  ENDIF
END SUBROUTINE Interact

SUBROUTINE Solve(E,F,G,Root1,Root2,IFail)
  IMPLICIT NONE
  REAL , INTENT(IN) :: E
  REAL , INTENT(IN) :: F
  REAL , INTENT(IN) :: G
  REAL , INTENT(OUT) :: Root1
  REAL , INTENT(OUT) :: Root2
  INTEGER , INTENT(INOUT) :: IFail
! Local variables
  REAL :: Term
  REAL :: A2
  Term = F*F - 4.*E*G
  A2 = E*2.0
! if term < 0, roots are complex
  IF(Term < 0.0)THEN
    IFail=1
  ELSE
    Term = SQRT(Term)
    Root1 = (-F+Term)/A2
    Root2 = (-F-Term)/A2
  ENDIF
END SUBROUTINE Solve

END PROGRAM ch2204
```

Thus in this chapter we have seen three ways of using subroutines:

- Classic Fortran 77 style as in the first example. The major disadvantage is the lack of checking of the parameters between the calling and called routine.
- Interface blocks — the major disadvantage is the code duplication
- Contained subroutines — major disadvantage is that we have to recompile the program and all contained subroutines.

We will look at using modules to address this problem in a later chapter.

22.5 Why bother?

Given the increase in the complexity of the overall program to solve a relatively straightforward problem, one must ask why bother. The answer lies in our ability to manage the solution of larger and larger problems. We need all the help we can get if we are to succeed in our task of developing large-scale reliable programs.

We need to be able to break our problems down into manageable subcomponents and solve each in turn. We are now in a very good position to be able to do this. Given a problem that requires a main program, one or more functions and one or more subroutines we can work on each subcomponent in relative isolation, and know that by using features like interface blocks we will be able to glue all of the components together into a stable structure at the end. We can independently compile the main program and functions and subroutines and use the linker to generate the overall executable, and then test that. Providing we keep our interfaces the same we can alter the actual implementations of the functions and subroutines and just recompile the changed procedures.

22.6 Summary

We now have the following concepts for the use of subroutines:

- INTERFACE blocks.
- INTENT attribute for parameters.
- Dummy parameters.
- The use of the CALL statement to invoke a subroutine.
- The concepts of variables that are local to the called routines and are unavailable elsewhere in the overall program.
- Communication between program units via the argument list.
- The concept of parameters on the call that enable us to report back on the status of the called routine.

22.7 Problems

1. Type in the program example in this chapter as three files. Compile each individually. When you have successfully compiled each routine (there will be the inevitable typing mistakes) look at the file sizes of the object file. Now use the linker to produce one executable. Look at the file size of the executable. What do you notice?

The development of large programs is eased considerably by the ability to compile small program units and eradicate the compilation errors from one unit at a time.

The linker obviously also has an important role to play in the development process.

2. Write a subroutine to calculate new coordinates (x', y') from (x, y) when the axes are rotated counterclockwise through an angle of a radians using:

$$x' = x \cos a + y \sin a$$

$$y' = -x \sin a + y \cos a$$

Hint:

The subroutine would look something like

SUBROUTINE ChangeCoordinate(X,Y,A,XD,YD)

Write a main program to read in values of *x,y,a*, call the subroutine and print out the new coordinates.

23
Subroutines: 2

"It is one thing to show a man he is in error, and another to put him in possession of the truth."

John Locke

Aims

The aims of this chapter are to extend the ideas in the earlier chapter on subroutines and look in more depth at parameter passing, in particular using a variety of ways of passing arrays.

23 Subroutines: 2

23.1 More on parameter passing

So far we have seen scalar parameters of type real, integer and logical. We will now look at numeric array parameters and character parameters. We need to introduce some technical terminology first. Don't panic if you don't fully understand the terminology the as examples should clarify things.

23.1.1 Explicit-shape array

An explicit-shape array is a named array that is declared with explicit values for the bounds in each dimension of the array.

The following explicit-shape arrays can specify nonconstant bounds:

- An automatic array (the array is a local variable).
- An adjustable array (the array is a dummy argument to a subprogram).

23.1.2 Assumed-shape array

An assumed-shape array is a nonpointer dummy argument array that takes its shape from the associated actual argument array.

23.1.3 Deferred-shape array

A deferred-shape array is an allocatable array or an array pointer. An allocatable array is an array that has the ALLOCATABLE attribute and a specified rank, but its bounds, and hence shape, are determined by allocation or argument association.

23.1.4 Automatic arrays

An automatic array is an explicit-shape array that is a local variable. Automatic arrays are only allowed in function and subroutine subprograms, and are declared in the specification part of the subprogram. At least one bound of an automatic array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

23.1.5 Assumed-size array — Fortran 77 style

An assumed-size array is a dummy argument array whose size is assumed from that of an associated actual argument. The rank and extents may differ for the actual and dummy arrays; only the size of the actual array is assumed by the dummy array. You would not use this type of parameter in modern Fortran code. We will come back to arrays of this type in the chapter on converting from Fortran 77 to modern Fortran.

23.1.6 Adjustable arrays — Fortran 77 style

An adjustable array is an explicit-shape array that is a dummy argument to a subprogram. At least one bound of an adjustable array must be a nonconstant specification expression. The bounds are determined when the subprogram is called. You would not use this type of parameter in modern Fortran code. We will come back to arrays of this type in the chapter on converting from Fortran 77 to modern Fortran.

23.2 Common code example

We are going to use an example based on a main program and a subroutine that calculates the mean and standard deviation of an array of numbers. The subroutine has the following parameters:

- x - the array containing the real numbers.
- n - the number of elements in the array.
- mean - the mean of the numbers.
- std_dev - the standard deviaition of the numbers.

We will look at some of the ways we can pass the array between the main program and the subroutine in both Fortran 77 and Fortran 90 styles.

23.3 Explicit-shape example

Consider the following program and subroutine.

```
program ch2301
implicit none
integer , parameter       :: n=10
real , dimension(1:n)     :: x
real , dimension(-4:5)    :: y
real , dimension(10)      :: z
real , allocatable , dimension(:) :: t
real :: m,sd
integer :: i

interface
  subroutine stats(x,n,mean,std_dev)
    implicit none
    integer , intent(in)                    :: n
    real     , intent(in) , dimension(1:n)  :: x
    real     , intent(out)                  :: mean
    real     , intent(out)                  :: std_dev
  end subroutine stats
end interface

  do i=1,n
    x(i)=real(i)
  end do
  call stats(x,n,m,sd)
  print *,' x'
  print *,' mean = ',m
  print *,' Standard deviation = ',sd
  y=x
  call stats(y,n,m,sd)
  print *,' y'
  print *,' mean = ',m
  print *,' Standard deviation = ',sd
  z=x
  call stats(z,10,m,sd)
  print *,' z'
  print *,' mean = ',m
  print *,' Standard deviation = ',sd
  allocate(t(10))
  t=x
  call stats(t,10,m,sd)
  print *,' t'
  print *,' mean = ',m
  print *,' Standard deviation = ',sd
end program ch2301

subroutine stats(x,n,mean,std_dev)
implicit none
integer , intent(in)                    :: n
real    , intent(in) , dimension(1:n) :: x
real    , intent(out)                   :: mean
real    , intent(out)                   :: std_dev
real :: variance
real:: sumxi,sumxi2
integer :: i

  variance=0.0
  sumxi=0.0
  sumxi2=0.0
  do i=1,n
```

```
      sumxi = sumxi+ x(i)
      sumxi2 = sumxi2 + x(i)*x(i)
   end do
   mean=sumxi/n
   variance = (sumxi2 - sumxi*sumxi/n)/(n-1)
   std_dev=sqrt(variance)
end subroutine stats
```

The key line in the subroutine is

```
real     , intent(in) , dimension(1:n) :: x
```

where the dummy array argument x is declared with explicit bounds and known as an explicit-shape dummy array. Even though it is not mandatory it is recommended that interface blocks be used so that the shape and size of actual and dummy arguments can be checked explicitly.

Note also the use of a DO loop to calculate the sum of the elements and the sum of the squares of the elements. This is a Fortran 77 style solution to this problem.

23.4 Assumed-shape example

A fundamental rule in modern Fortran is that the shape of an actual array argument and its associated dummy arguments are the same, i.e., they both must have the same rank and the same extents in each dimension. The best way to apply this rule is to use assumed-shape dummy array arguments as shown in the example below.

In the subroutine we have

```
real , intent(in) , dimension(:) :: x
```

where x is an assumed-shape dummy array argument, and it will assume the shape of the actual argument when the subroutine is called.

In this example in the main program we have declared the actual array argument x to be allocatable to make the program more flexible.

```
program ch2302

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

interface
   subroutine stats(x,n,mean,std_dev)
```

```
    implicit none
    integer , intent(in)                    :: n
    real    , intent(in) , dimension(:) :: x
    real    , intent(out)                   :: mean
    real    , intent(out)                   :: std_dev
  end subroutine stats
end interface

  print *,' type in n'
  read *,n
  allocate(x(1:n))
  call random_number(x)
  x=x*100
  call stats(x,n,m,sd)
  print *,' numbers were '
  print *,x
  print *,' Mean =                 ',m
  print *,' Standard deviation = ',sd

end program ch2302

subroutine stats(x,n,mean,std_dev)
implicit none
integer , intent(in)                    :: n
real    , intent(in) , dimension(:)    :: x
real    , intent(out)                   :: mean
real    , intent(out)                   :: std_dev
real :: variance
real:: sumxi,sumxi2
integer :: i

  variance=0.0
  sumxi=0.0
  sumxi2=0.0
  do i=1,n
     sumxi  = sumxi+ x(i)
     sumxi2 = sumxi2 + x(i)*x(i)
  end do
  mean=sumxi/n
  variance = (sumxi2 - sumxi*sumxi/n)/(n-1)
  std_dev=sqrt(variance)
end subroutine stats
```

23.4.1 Notes

There are several restrictions when using assumed-shape arrays:

- The rank is equal to the number of colons, in this case 1.
- The lower bounds of the assumed-shape array are the specified lower bounds, if present, and 1 otherwise. In the example above it is 1 because we haven't specified a lower bound.
- The upper bounds will be determined on entry to the procedure and will be whatever values are needed to make sure that the extents along each dimension of the dummy argument are the same as the actual argument. In this case the upper bound will be n.
- An assumed-shape array must not be defined with the POINTER or ALLOCATABLE attribute in Fortran 90 or Fortran 95.
- When using an assumed-shape array an interface block is mandatory.

Assumed-shape arraay parameter passing also works with Fortran 77 style statically allocated arrays, i.e.,

```
real , dimension(1:10) :: x
```

which is commonly seen in older code.

23.5 Character arguments and assumed-length dummy arguments

The types of parameters considered so far have been REAL, INTEGER and LOGICAL. CHARACTER variables are slightly different because they have a length associated with them. Consider the following program and subroutine which, given the name of a file, opens it and reads values into two REAL arrays, X and Y:

```
PROGRAM ch2303
IMPLICIT NONE
REAL,DIMENSION(1:100)::A,B
INTEGER :: Nos,I
CHARACTER(LEN=20)::Filename
INTERFACE
   SUBROUTINE Readin(Name,X,Y,N)
      IMPLICIT NONE
      INTEGER , INTENT(IN) :: N
      REAL,DIMENSION(1:N),INTENT(OUT)::X,Y
      CHARACTER (LEN=*),INTENT(IN)::Name
   END SUBROUTINE Readin
END INTERFACE
   PRINT *,' Type in the name of the data file'
   READ '(A)' , Filename
   PRINT *,' Input the number of items in the file'
   READ * , Nos
   CALL Readin(Filename,A,B,Nos)
   PRINT * , ' Data read in was'
   DO I=1,Nos
      PRINT *,' ',A(I),' ',B(I)
   ENDDO

END PROGRAM ch2303

SUBROUTINE Readin(Name,X,Y,N)
IMPLICIT NONE
INTEGER , INTENT(IN) :: N
REAL,DIMENSION(1:N),INTENT(OUT)::X,Y
CHARACTER (LEN=*),INTENT(IN)::Name
INTEGER::I
   OPEN(UNIT=10,STATUS='OLD',FILE=Name)
   DO I=1,N
      READ(10,*)X(I),Y(I)
   END DO
   CLOSE(UNIT=10)
END SUBROUTINE Readin
```

The main program reads the file name from the user and passes it to the subroutine that reads in the data. The dummy argument Name is of type assumed-length, and picks up the length from the actual argument Filename in the calling routine, which is in this case 20 characters. An interface block **must** be used with assumed-shape dummy arguments.

23.6 Rank 2 and higher arrays as parameters

23.6.1 Explicit-shape dummy arrays

Consider the following example which uses a Fortran 77 style of two-dimensional array parameter passing.

In the main program we have the following declaration of the rank 2 actual array arguments:

```
REAL   , DIMENSION (1:Max,1:Max)::One,Two,Three,One_T
```

and in the subroutine Matrix_bits we declare the rank 2 dummy array arguments as follows:

```
  REAL, DIMENSION (1:Max,1:Max), INTENT(IN) :: A,B
  REAL, DIMENSION (1:Max,1:Max), INTENT(OUT) :: C,A_T
```

We have split the declaration into two as the arrays have different intents. These dummy array arguments are explicit-shape, i.e., their bounds are declared in the subroutine.

Note that in the main program N may be less than Max and because of the way Fortran stores arrays internally we must pass both variables as arguments to the subroutine Matrix_bits, N being used to control the DO loops and Max needed in the array declarations:

```
PROGRAM ch2304
IMPLICIT NONE
INTEGER, PARAMETER :: Max=10
REAL   , DIMENSION (1:Max,1:Max)::One,Two,Three,One_T
INTEGER :: I,N
INTERFACE
  SUBROUTINE Matrix_bits(A,B,C,A_T,N,Max)
  IMPLICIT NONE
  INTEGER, INTENT(IN):: N, Max
  REAL, DIMENSION (1:Max,1:Max), INTENT(IN) :: A,B
  REAL, DIMENSION (1:Max,1:Max), INTENT(OUT) :: C,A_T
  END SUBROUTINE Matrix_bits
END INTERFACE

    PRINT *,'Input size of matrices'
    READ*,N
    DO WHILE(N > Max)
      PRINT*,'size of matrices must be <= ',Max
      PRINT *,'Input size of matrices'
    READ*,N
    END DO
    DO I=1,N
      PRINT*, 'Input row ', I,' of One'
      READ*,One(I,1:N)
    END DO
    DO I=1,N
      PRINT*, 'Input row ', I,' of Two'
      READ*,Two(I,1:N)
    END DO
    CALL Matrix_bits(One,Two,Three,One_T,N,Max)
    PRINT*,' Matrix Three:'
    DO I=1,N
      PRINT *,Three(I,1:N)
    END DO
    PRINT *,' Matrix One_T:'
    DO I=1,N
      PRINT *,One_T(I,1:N)
    END DO
END PROGRAM ch2304

SUBROUTINE Matrix_bits(A,B,C,A_T,N,Max)
IMPLICIT NONE
INTEGER, INTENT(IN):: N, Max
REAL, DIMENSION (1:Max,1:Max), INTENT(IN) :: A,B
REAL, DIMENSION (1:Max,1:Max), INTENT(OUT) :: C,A_T

INTEGER::I,J,K
REAL:: Temp
!
! matrix multiplication C=A B
!
   DO I=1,N
      DO J=1,N
         Temp=0.0
         DO K=1,N
            Temp = Temp + A(I,K) * B (K,J)
         END DO
            C(I,J) = Temp
      END DO
   END DO
!
! set A_T to be transpose matrix A
  DO I=1,N
     DO J=1,N
        A_T(I,J) = A(J,I)
     END DO
 END DO
END SUBROUTINE Matrix_bits
```

Note the use of DO loops to carry out the matrix multiplication and transpose. This is a Fortran 77 style solution to the problem.

23.6.2 Assumed-shape dummy array arguments

With the introduction of assumed-shape dummy array arguments the necessity to pass through Max in the last program is removed. This is shown in the example below:

```
PROGRAM ch2305
  IMPLICIT NONE
  REAL , ALLOCATABLE , DIMENSION &
  (:,:)::One,Two,Three,One_T
  INTEGER :: I,N
  INTERFACE
    SUBROUTINE Matrix_bits(A,B,C,A_T,N)
    IMPLICIT NONE
    INTEGER, INTENT(IN):: N
      REAL, DIMENSION (:,:), INTENT(IN) :: A,B
      REAL, DIMENSION (:,:), INTENT(OUT) :: C,A_T
    END SUBROUTINE Matrix_bits
  END INTERFACE
    PRINT *,'Input size of matrices'
    READ*,N
    ALLOCATE(One(1:N,1:N))
    ALLOCATE(Two(1:N,1:N))
    ALLOCATE(Three(1:N,1:N))
    ALLOCATE(One_T(1:N,1:N))
    DO I=1,N
      PRINT*, 'Input row ', I,' of One'
      READ*,One(I,1:N)
    END DO
    DO I=1,N
      PRINT*, 'Input row ', I,' of Two'
      READ*,Two(I,1:N)
    END DO
    CALL Matrix_bits(One,Two,Three,One_T,N)
    PRINT*,' Matrix Three:'
    DO I=1,N
      PRINT *,Three(I,1:N)
    END DO
    PRINT *,' Matrix One_T:'
    DO I=1,N
      PRINT *,One_T(I,1:N)
    END DO
END PROGRAM ch2305
```

```
SUBROUTINE Matrix_bits(A,B,C,A_T,N)
  IMPLICIT NONE
  INTEGER, INTENT(IN):: N
  REAL, DIMENSION (:,:), INTENT(IN) :: A,B
  REAL, DIMENSION (:,:), INTENT(OUT) :: C,A_T
  INTEGER:: I,J, K
  REAL:: Temp
!
! matrix multiplication C=AB
!
   DO I=1,N
      DO J=1,N
         Temp=0.0
         DO K=1,N
            Temp = Temp + A(I,K) * B (K,J)
         END DO
            C(I,J) = Temp
      END DO
   END DO
!
! Calculate A_T transpose of A
!
!
! set A_T to be transpose matrix A
  DO I=1,N
     DO J=1,N
        A_T(I,J) = A(J,I)
     END DO
 END DO
END SUBROUTINE Matrix_bits
```

23.6.3 Notes

The dummy array and actual array arguments look the same but there is a difference:

- The dummy array arguments A, B, C, A_T are all assumed-shape arrays and take the shape of the actual array arguments One, Two, Three and One_T, respectively.
- The actual array arguments One, Two, Three and One_T in the main program are allocatable arrays or deferred-shape arrays. An allocatable array

is an array that has an allocatable attribute. Its bounds and shape are declared when the array is allocated, hence deferred-shape.

23.6.4 Using the intrinsic functions MATMUL and TRANSPOSE

In the previous two examples the matrix multiplication and transpose were hand coded, and are what you would see in Fortran 77 style code. This example uses the built in intrinsics MATMUL and TRANSPOSE and is modern Fortran 90 style:

```
PROGRAM ch2306
  IMPLICIT NONE
  REAL , ALLOCATABLE , DIMENSION &
  (:,:)::One,Two,Three,One_T
  INTEGER :: I,N
  INTERFACE
    SUBROUTINE Matrix_bits(A,B,C,A_T)
      IMPLICIT NONE
      REAL, DIMENSION (:,:), INTENT(IN) :: A,B
      REAL, DIMENSION (:,:), INTENT(OUT) :: C,A_T
    END SUBROUTINE Matrix_bits
  END INTERFACE
    PRINT *,'Input size of matrices'
    READ*,N
    ALLOCATE(One(1:N,1:N))
    ALLOCATE(Two(1:N,1:N))
    ALLOCATE(Three(1:N,1:N))
    ALLOCATE(One_T(1:N,1:N))
    DO I=1,N
      PRINT*, 'Input row ', I,' of One'
      READ*,One(I,1:N)
    END DO
    DO I=1,N
      PRINT*, 'Input row ', I,' of Two'
      READ*,Two(I,1:N)
    END DO
    CALL Matrix_bits(One,Two,Three,One_T)
    PRINT*,' Matrix Three:'
    DO I=1,N
      PRINT *,Three(I,1:N)
    END DO
    PRINT *,' Matrix One_T:'
    DO I=1,N
      PRINT *,One_T(I,1:N)
    END DO
END PROGRAM ch2306

SUBROUTINE Matrix_bits(A,B,C,A_T)
   IMPLICIT NONE
   REAL, DIMENSION (:,:), INTENT(IN) :: A,B
   REAL, DIMENSION (:,:), INTENT(OUT) :: C,A_T
      C=MATMUL(A,B)
      A_T=TRANSPOSE(A)
END SUBROUTINE Matrix_bits
```

Fortran thus provides a variety of ways of passing array parameters. We have covered both 77 and 90 styles, as you will see both in code that you work with.

23.7 Automatic arrays and median calculation

This example looks at the calculation of the median of a set of numbers and also illustrates the use of an automatic array.

The median is the middle value of a list, i.e., the smallest number such that at least half the numbers in the list are no greater. If the list has an odd number of entries, the median is the middle entry in the list after sorting the list into ascending order. If the list has an even number of entries, the median is equal to the sum of the two middle (after sorting) numbers divided by two. One way to determine the median computationally is to sort the numbers and choose the item in the middle.

Wirth classifies sorting into simple and advanced, and his three simple methods are as follows:

- Insertion sorting — The items are considered one at a time and each new item is inserted into the appropriate position relative to the previously sorted item. If you have ever played bridge then you have probably used this method.
- Selection sorting — First the smallest (or largest) item is chosen and is set aside from the rest. Then the process is repeated for the next smallest item and set aside in the next position. This process is repeated until all items are sorted.
- Exchange sorting — If two items are found to be out of order they are interchanged. This process is repeated until no more exchanges take place.

Knuth also identifies the above three sorting methods. For more information on sorting the Knuth and Wirth books are good starting places. Knuth is a little old (1973) compared to Wirth (1986), but it is still a very good coverage. Knuth uses mix assembler to code the examples whilst the Wirth book uses Modula 2, and is therefore easier to translate into modern Fortran.

In the example below we use a selection sort:

```
program ch2307

implicit none
integer :: n
real , allocatable , dimension(:) :: x
real :: m,sd,median

interface
  subroutine stats(x,n,mean,std_dev,median)
    implicit none
    integer , intent(in)                   :: n
    real     , intent(in) , dimension(:)   :: x
    real     , intent(out)                 :: mean
    real     , intent(out)                 :: std_dev
    real     , intent(out)                 :: median
  end subroutine stats
end interface

  print *,' How many values ?'
  read *,n
  allocate(x(1:n))
  call random_number(x)
  x=x*1000
  call stats(x,n,m,sd,median)
  print *,' mean = ',m
  print *,' Standard deviation = ',sd
  print *,' median is = ',median

end program ch2307

subroutine stats(x,n,mean,std_dev,median)
implicit none
integer , intent(in)                   :: n
real     , intent(in) , dimension(:)   :: x
real     , intent(out)                 :: mean
real     , intent(out)                 :: std_dev
real     , intent(out)                 :: median
real     , dimension(1:n)             :: y
real :: variance
real     :: sumxi, sumxi2
  sumxi=0.0
```

```
   sumxi2=0.0
   variance=0.0
   sumxi=sum(x)
   sumxi2=sum(x*x)
   mean=sumxi/n
   variance=(sumxi2-sumxi*sumxi/n)/(n-1)
    std_dev = sqrt(variance)
   y=x
   call selection
   if (mod(n,2) == 0) then
     median=(y(n/2)+y((n/2)+1))/2
   else
     median=y((n/2)+1)
   endif
contains

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

end subroutine stats
```

In the subroutine stats the array y is automatic. It will be allocated automatically when we call the subroutine. We use this array as a work array to hold the sorted data. We then use this sorted array to determine the median.

Note the use of the SUM intrinsic in this example:

```
  sumxi=sum(x)
  sumxi2=sum(x*x)
```

These statements replace the DO loop from the earlier example. A good optimising compiler would not make two passes over the data with these two statements.

23.7.1 Internal subroutines and scope

The stats subroutine contains the selection subroutine. The stats subroutine has access to the following variables

- x,n,mean,std_dev, median — these are made available as they are passed in as parameters.
- y, variance, sumxi, sumxi2 — are local to the subroutine stats.

The subroutine selection has access to the above as it is contained within subroutine stats. It also has the following local variables that are only available within subroutine selection

- i,j,k, minimum

23.7.2 Timing the selection sort algorithm

The selection sort is a simple algorithm and the following main program illustrates it limitations with increasing n. It uses the same stats subroutine as the previous example:

```
program ch2308

implicit none
integer :: n
real , allocatable , dimension(:) :: x
real :: m,sd,median
integer , dimension(8) :: timing

interface
  subroutine stats(x,n,mean,std_dev,median)
    implicit none
    integer , intent(in)                     :: n
    real     , intent(in) , dimension(:)    :: x
    real     , intent(out)                   :: mean
    real     , intent(out)                   ::
std_dev
    real     , intent(out)                   :: median
  end subroutine stats
end interface
```

```
  n=1000
  do
    print *,' n = ',n
    allocate(x(1:n))
    call random_number(x)
    x=x*1000
    call date_and_time(values=timing)
    print *,' initial '
    print *, timing(6) , timing(7) , timing(8)
    call stats(x,n,m,sd,median)
    print *,' mean = ',m
    print *,' Standard deviation = ',sd
    print *,' median is = ',median
    call date_and_time(values=timing)
    print *,' after'
    print *,timing(6),timing(7),timing(8)
    n=n*10
    deallocate(x)
  end do

end program ch2308
```

23.7.2.1 Timing

Dell Precision Workstation, 2 * 933 MHz, 512 Mb ram:

```
  n =  1000
  initial                                        9 13 906
  mean =     5.0895782E+02
  Standard deviation =     2.8708249E+02
  median is =     5.1872925E+02
  after sort                                    9 13 950
  n =  10000
  initial                                        9 13 951
  mean =     4.9967194E+02
  Standard deviation =     2.8635922E+02
  median is =     5.0259839E+02
  after sort                                    9 14 689
  n =  100000
  initial                                        9 14 697
  mean =     5.0123392E+02
  Standard deviation =     2.8869482E+02
  median is =     4.9957404E+02
  after sort                                      11 12 292
```

Dell Inspiron, 1 * 3.4 Ghz, 1 Gb ram:

```
  n =  1000
  initial                                          10 57 421
  mean =     5.0781586E+02
  Standard deviation =     2.9026807E+02
  median is =     5.1530060E+02
  after sort                                      10 57 524
  n =  10000
  initial                                          10 57 525
  mean =     4.9770724E+02
  Standard deviation =     2.8532513E+02
  median is =     4.9151331E+02
  after sort                                      10 57 778
  n =  100000
  initial                                          10 57 781
  mean =     4.9930457E+02
  Standard deviation =     2.8866571E+02
  median is =     4.9931268E+02
  after sort                                      11 23 374
```

This algorithm is approximately order n * log(n).

23.8 Alternative median calculation algorithm

This program uses an algorithm developed by Hoare to determine the median. The number of computations required to find the median is approximately 2 * n.

Timings are given at the end:

```
program ch2309

implicit none
integer :: n
real , allocatable , dimension(:) :: x
real :: m,sd,median
integer , dimension(8) :: timing

interface
  subroutine stats(x,n,mean,std_dev,median)
    implicit none
    integer , intent(in)                :: n
    real     , intent(in) , dimension(:)    :: x
    real     , intent(out)                  :: mean
    real     , intent(out)                  :: std_dev
    real     , intent(out)                  :: median
  end subroutine stats
end interface

  n=1000
  do
    print *,' n = ',n
    allocate(x(1:n))
    call random_number(x)
    x=x*1000
    call date_and_time(values=timing)
    print *,' initial '
    print *,timing(6),timing(7),timing(8)
    call stats(x,n,m,sd,median)
    print *,' mean = ',m
    print *,' Standard deviation = ',sd
    print *,' median is = ',median
    call date_and_time(values=timing)
    print *,' after sort'
    print *, timing(6),timing(7),timing(8)
    n=n*10
    deallocate(x)
  end do

end program ch2309

subroutine stats(x,n,mean,std_dev,median)
implicit none
integer , intent(in)                    :: n
real     , intent(in) , dimension(:)    :: x
real     , intent(out)                  :: mean
real     , intent(out)                  :: std_dev
real     , intent(out)                  :: median
real     , dimension(1:n)              :: y
real :: variance
real     :: sumxi, sumxi2
integer:: k
  sumxi=0.0
  sumxi2=0.0
```

```
variance=0.0
sumxi=sum(x)
sumxi2=sum(x*x)
mean=sumxi/n
variance=(sumxi2-sumxi*sumxi/n)/(n-1)
std_dev = sqrt(variance)
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 subroutine stats
```

23.8.1 Timing

Dell Precision Workstation, 2 * 933 MHz, 512 Mb ram:

```
  n =  1000
  initial                                   19 37 421
  mean =    4.9430524E+02
  Standard deviation =    2.9318149E+02
  median is =    4.8815854E+02
  after sort                               19 37 426
  n =  10000
  initial                                   19 37 427
  mean =    4.9803854E+02
  Standard deviation =    2.9065613E+02
  median is =    4.9482861E+02
  after sort                               19 37 431
  n =  100000
  initial                                   19 37 439
  mean =    5.0035132E+02
  Standard deviation =    2.8867920E+02
  median is =    5.0099771E+02
  after sort                               19 37 468
  n =  1000000
  initial                                   19 37 542
  mean =    4.9944907E+02
  Standard deviation =    2.8857736E+02
  median is =    4.9952847E+02
  after sort                               19 37 838
  n =  10000000
  initial                                   19 38 626
  mean =    4.9974246E+02
  Standard deviation =    2.8268231E+02
```

```
 median is =    4.9996432E+02
 after sort                                19 41 722
```

Dell Inspiron, 1 * 3.4 Ghz, 1 Gb ram:

```
 n =  1000
 initial                                    21 38 734
 mean =    4.9351086E+02
 Standard deviation =    2.9076068E+02
 median is =    4.8678186E+02
 after sort                                21 38 734
 n =  10000
 initial                                    21 38 734
 mean =    5.0000485E+02
 Standard deviation =    2.8909946E+02
 median is =    4.9666431E+02
 after sort                                21 38 765
 n =  100000
 initial                                    21 38 768
 mean =    5.0053433E+02
 Standard deviation =    2.8863885E+02
 median is =    4.9899475E+02
 after sort                                21 38 775
 n =  1000000
 initial                                    21 38 797
 mean =    5.0039590E+02
 Standard deviation =    2.8852356E+02
 median is =    5.0053967E+02
 after sort                                21 38 855
 n =  10000000
 initial                                    21 39 67
 mean =    4.9973923E+02
 Standard deviation =    2.8260712E+02
 median is =    4.9987134E+02
 after sort                                21 39 806
 n =  100000000
 initial                                    21 41 930
 mean =    1.7179869E+02
 Standard deviation =    3.8263177E+02
 median is =    5.0002957E+02
 after sort                                21 59 222
```

The differences between the two algorithms and systems are summarised below:

System	N	Selection	Find
Dual	100,000,000	NA	NA
	10,000,000	NA	3.096
	1,000,000	NA	0.296
	100,000	117.595	0.029
	10,000	0.738	0.004
Single	100,000,000	NA	17.292
	10,000,000	NA	0.739
	1,000,000	NA	0.058
	100,000	25.593	NA
	10,000	0.253	NA

Hoare's Find algorithm is obviously much faster, but far less easy to understand than the simple selection sort.

The limiting factor with this algorithm on these systems is the amount of installed memory. The program crashes on both systems with a failure to allocate the automatic array. This is a drawback of automatic arrays in that there is no mechanism to handle this failure gracefully.You would then need to use allocatable local work arrays. The drawback here is that the programmer is then responsible for the deallocation of these arrays. Memory leaks are then possible.

23.9 Recursive subroutines — Quicksort

In Chapter 14 we saw an example of recursive functions. This example illustrates the use of recursive subroutines. It uses a simple implementation of Hoare's Quicksort. References are given in the bibliography. The overall problem is broken down into:

- A main program that prompts the user for the name of the data file and n. The allocation of the array is carried out in the main program.
- A subroutine to read the data.
- A subroutine to sort the data. This subroutine contains the recursive subroutine Quicksort.
- A subroutine to write the sorted data to a file.

Below is the complete program:

```
PROGRAM ch2310
IMPLICIT NONE
INTEGER                                   :: How_Many
CHARACTER   (LEN=20)                    :: File_Name
REAL , ALLOCATABLE , DIMENSION(:)   :: Raw_Data
integer , dimension(8) :: timing

INTERFACE
  SUBROUTINE Read_Data(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
  END SUBROUTINE Read_Data
END INTERFACE

INTERFACE
  SUBROUTINE Sort_Data(Raw_Data,How_Many)
    IMPLICIT NONE
    INTEGER , INTENT(IN) :: How_Many
    REAL , INTENT(INOUT) , &
    DIMENSION(:) :: Raw_Data
  END SUBROUTINE Sort_Data
END INTERFACE

INTERFACE
  SUBROUTINE Print_Data(Raw_Data,How_Many)
    IMPLICIT NONE
    INTEGER , INTENT(IN) :: How_Many
    REAL , INTENT(IN) , &
    DIMENSION(:) :: Raw_Data
  END SUBROUTINE Print_Data
END INTERFACE

  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=timing)
  print *,' initial'
  print *,timing(6),timing(7),timing(8)
  ALLOCATE(Raw_Data(How_Many))
  call date_and_time(values=timing)
  print *,' allocate'
  print *,timing(6),timing(7),timing(8)
  CALL Read_Data(File_Name,Raw_Data,How_Many)
  call date_and_time(values=timing)
  print *,' read'
  print *,timing(6),timing(7),timing(8)
  CALL Sort_Data(Raw_Data,How_Many)
  call date_and_time(values=timing)
  print *,' sort'
  print *,timing(6),timing(7),timing(8)
  CALL Print_Data(Raw_Data,How_Many)
  call date_and_time(values=timing)
  print *,' print'
  print *,timing(6),timing(7),timing(8)
  PRINT * , ' '
  PRINT *, ' Data written to file SORTED.DAT'

END PROGRAM ch2310

SUBROUTINE Read_Data(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
! Local variables
INTEGER :: I

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

END SUBROUTINE Read_Data

SUBROUTINE Sort_Data(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
  ! Local variables
  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_Data
```

```
SUBROUTINE Print_Data(Raw_Data,How_Many)
IMPLICIT NONE
INTEGER , INTENT(IN) :: How_Many
REAL , INTENT(IN) , DIMENSION(:) :: Raw_Data
! Local variables
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_Data
```

23.9.1 Note — Interface blocks

We introduced interface blocks in Chapter 22, and in that chapter the parameters were scalars; in this example we have a mix of arrays and scalars. The above program is in Fortran 77 style with a main program and several distinct subroutines. The use of interface blocks is recommended when using this style of programming as it will minimise cross program unit (program, function, subroutine) errors.

Interface blocks are **mandatory** under the following situations:

- The procedure has optional arguments.
- When a function returns an array.
- When a function returns a pointer.
- For character functions a result that is dynamic.
- When the procedure has assumed-shape dummy arguments.
- When the procedure has dummy arguments with the pointer attribute.
- When the procedure has dummy arguments with the target attribute.
- When the procedure has keyword arguments and/or optional arguments.
- When the procedure is generic. You have already seen that some of the intrinsic procedures have this status, e.g., SINE will return a result when the argument is of a variety of types. This means that we can construct procedures that will accept arguments of a variety of types and all we need to do is provide a procedure that manipulates data of that type. We will look at the construction of a procedure that is generic in a later chapter.
- When the procedure provides a user defined operator.

- When the procedure provides user defined assignment.

23.9.2 Note — Recursive subroutine

The actual sorting is done in the recursive subroutine `QuickSort.` The actual algorithm is taken from the Wirth book. See the bibliography for a reference.

Recursion provides us with a very clean and expressive way of solving many problems. There will be instances where it is worthwhile removing the overhead of recursion, but the first priority is the production of a program that is correct. It is pointless having a very efficient but incorrect solution.

We will look again at recursion and efficiency in a later chapter and see under what criteria we can replace recursion with iteration.

23.9.3 Note — Flexible design

The QuickSort recursive routine can be replaced with another sorting algorithm and we can maintain the interface to Sort_Data. We can thus decouple the implementation of the actual sorting routine from the defined interface. We would only need to recompile the Sort_Data routine and we could relink using the already compiled main, read data and print data routines.

23.9.4 Note — Timing information

We call the date_and_time intrinsic subroutine to get timing information. A summary table from runnning the above program on a 3.4 Ghz system with 1 Gb memory with four different compilers is given below:

		n=10,000,000	
Compiler			
	read	sort	print
1	7.328	2.812	61.391
2	7.563	2.922	58.843
3	5.765	3.281	17.751
4	7.015	3.438	22.344

As can be seen it is the I/O that dominates the overall running time of the program. In the 10 years since first running this program we have seen the data set size increase from tens of thousands to tens and hundreds of millions.

23.10 Summary

We now have a lot of the tools to start tackling problems in a structured and modular way, breaking problems down into manageable chunks and designing subprograms for each of the tasks.

23.11 Problems

1. Below is the random number program that was used to generate the data sets for the Quicksort example:

```
program ch2311
implicit none
integer :: n
integer :: i
real , allocatable , dimension(:) :: x
  print *,' how many values ?'
  read *,n
  allocate(x(1:n))
  call random_number(x)
  x=x*1000
  open(unit=10,file='random.txt')
  do i=1,n
    write(10, 100)x(i)
    100 format(f8.3)
  end do
end program ch2311
```

Run the Quick_Sort program in this chapter with the data file as input. Obtain timing details.

What percentage of the time does the program spend in each subroutine? Is it worth trying to make the sort much more efficient given these timings?

2. Find out if there is a subroutine library like the NAG library available. If there is replace the Quick_Sort recursive subroutine with a suitable routine from that library. What times do you obtain?

3. Try using the operating system SORT command to sort the file. What timing figures do you get now?

Was it worth writing a program?

4. Consider the following program:

```
program ch2312
!
! Program to test array subscript checking
! when the array is passed as an argument.
!
implicit none
integer , parameter :: array_size=10
integer :: i
integer , dimension(array_size) :: a
  do i=1,array_size
    a(i)=i
  end do
  call sub01(a,array_size)
end program ch2312

subroutine sub01(a,array_size)
implicit none
integer , intent(in) :: array_size
integer , intent(in) , dimension(array_size) :: a
integer :: i
integer :: atotal=0
integer :: rtotal=0
  do i=1,array_size
    rtotal=rtotal+a(i)
  end do
  do i=1,array_size+1
    atotal=atotal+a(i)
  end do
  print *,' Apparent total is ' , atotal
  print *,'      real total is ' , rtotal
end subroutine sub01
```

The key thing to note is that we haven't used interface blocks and we have an error in the subroutine where we go outside the array. Run this program. What answer do you get for the apparent total?

Are there any compiler flags or switches which will enable you to trap this error?

23.12 Bibliography

Hoare C.A.R., *Algorithm 63, Partition; Algorithm 64, Quicksort, p.321; Algorithm 65: FIND*, Comm. of the ACM, 4 p.321–322, 1961.

Hoare C.A.R., *Proof of a Program: FIND*, Comm A.C.M., 13, No 1 (1970) 39–45

Hoare C.A.R., *Proof of a Recursive Program: Quicksort*, Comp. J., 14, No 4 (1971) 391–95.

Knuth D.E., *The Art of Computer Programming*, Volume 3 — *Sorting and Searching*, Addison-Wesley, 1973.

Wirth N., *Algorithms and Data Structures*, Prentice-Hall, 1986.

23.13 Commercial numerical and statistical subroutine libraries

There are two major suppliers of commercial libraries:

- NAG: Numerical Algorithms Group

and

- Visual Numerics

They can be found at:

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

and

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

respectively. Their libraries are written by numerical analysts, and are fully tested and well documented. They are under constant development and available for a wide range of hardware platforms and compilers. Parallel versions are also available.