Chapter 20
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.

20.1 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 as the examples should clarify things.

20.1.1 Assumed-Shape Array

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

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_20

20.1.2 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.

20.1.3 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.

20.1.4 Allocatable Dummy Arrays

Fortran provides the ability to declare an array in the main program and allocate in a subroutine.

20.1.5 Keyword and Optional Arguments

Fortran provides the ability to supply the actual arguments to a procedure by keyword, and hence in any order.

To do this the name of the dummy argument is referred to as the keyword and is specified in the actual argument list in the form

```
dummy-argument = actual-argument
```

A number of points need to be noted when using keyword and optional arguments:

- if all the actual arguments use keywords, they may appear in any order.
- When only some of the actual arguments use keywords, the first part of the list must be positional followed by keyword arguments in any order.
- When using a mixture of positional and keyword arguments, once a keyword argument is used all subsequent arguments must be specified by keyword.
- if an actual argument is omitted the corresponding optional dummy argument must not be redefined or referenced, except as an argument to the `present` intrinsic function.

- if an optional dummy argument is at the end of the argument list then it can just be omitted from the actual argument list.
- Keyword arguments are needed when an optional argument not at the end of an argument list is omitted, unless all the remaining arguments are omitted as well.
- Keyword and optional arguments require explicit procedure interfaces, i.e., the procedure must be internal, a module procedure or have an interface block available in the calling program unit.

A number of the intrinsic procedures have optional arguments. Consult Appendix D for details. We look at a complete example using optional arguments in a later chapter.

20.2 Example 1: Assumed Shape Parameter Passing

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 deviation of the numbers.

Consider the following program and subroutine.

```
module statistics_module
  implicit none

contains
  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
end module statistics_module

program ch2001
  use statistics_module
  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

  do i = 1, n
    x(i) = real(i)
  end do
  call stats(x, n, m, sd)
  print *, ' x'
  print 100, m, sd
100 format (' Mean = ', f7.3, ' Std Dev = ', &
    f7.3)
  y = x
  call stats(y, n, m, sd)
  print *, ' y'
  print 100, m, sd
  z = x
  call stats(z, 10, m, sd)
  print *, ' z'
  print 100, m, sd
  allocate (t(n))
  t = x
  call stats(t, 10, m, sd)
  print *, ' t'
  print 100, m, sd
end program ch2001
```

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 above.

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 two of the calls we have passed a variable `n` as the size of the array and used a literal integer constant (10) in the other two cases. Both parameter passing mechanisms work.

20.2.1 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.
- When using an assumed-shape array an interface is mandatory. In this example it is provided by the the `stats` subroutine being a contained subroutine in a module, and the use of the module in the main program.

20.3 Example 2: 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 the real array x:

```
module read_module
  implicit none

contains
  subroutine readin(name, x, n)
```

```
    implicit none
    integer, intent (in) :: n
    real, dimension (:), intent (out) :: x
    character (len=*), intent (in) :: name
    integer :: i

    open (unit=10, file=name, status='old')
    do i = 1, n
      read (10, *) x(i)
    end do
    close (unit=10)
  end subroutine readin
end module read_module

program ch2002
  use read_module
  implicit none
  real, allocatable, dimension (:) :: a
  integer :: nos, i
  character (len=20) :: filename

  print *, ' Type in the name of the data file'
  read '(a)', filename
  print *, ' Input the number of items'
  read *, nos
  allocate (a(1:nos))
  call readin(filename, a, nos)
  print *, ' data read in was'
  do i = 1, nos
    print *, ' ', a(i)
  end do
end program ch2002
```

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 must be provided with assumed-shape dummy arguments, and this is achieved in this case by the subroutine being in a module.

20.4 Example 3: Rank 2 and Higher Arrays as Parameters

The following example illustrates the modern way of passing rank 2 and higher arrays as parameters. We start with a simple rank 2 example.

```
module matrix_module
  implicit none

contains
  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
end module matrix_module

program ch2003
  use matrix_module
  implicit none
  real, allocatable, dimension (:, :) :: one, &
    two, three, one_t
  integer :: i, n
```

```
    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 ch2003
```

The subroutine is doing a matrix multiplication and transpose. There are intrinsic functions in Fortran called `matmul` and `transpose` that provide the same functionality as the subroutine. One of the problems at the end of the chapter is to replace the code in the subroutine with calls to the intrinsic functions.

20.4.1 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.

20.5 Example 4: 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 an exchange sort:

```
module statistics_module
  implicit none

contains
  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
  if (mod(n,2)==0) then
    median = (find(n/2)+find((n/2)+1))/2
  else
    median = find((n/2)+1)
  end if
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
end module statistics_module

program ch2004
  use statistics_module
  implicit none
  integer :: n
  integer :: i
  real, allocatable, dimension (:) :: x
  real :: m, sd, median
  integer, dimension (8) :: timing

  n = 1000000
  do i = 1, 3
    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 *, timing(6), timing(7), timing(8)
    n = n*10
    deallocate (x)
  end do
end program ch2004
```

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.

20.5.1 Internal Subroutines and Scope

The `stats` subroutine contains the `find` 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 `find` 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`

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.

The limiting factor with this algorithm is the amount of installed memory. The program will crash on 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.

20.6 Example 5: Recursive Subroutines – Quicksort

In Chap. 12 we saw an example of recursive functions. This example illustrates the use of a recursive subroutine. In this example we use the additional form of the subroutine header that was required when recursive procedure support was introduced in Fortran 90. The Fortran 2018 standard makes this form optional. It uses a simple implementation of Hoare's Quicksort. References are given in the bibliography. We took the algorithm from Wirth's book for our example.

The program times the various components parts of the program

- dynamic allocation of the real array
- use the `random_number` subroutine to generate the numbers
- call the `sort_data` subroutine to sort the data
- print out the first 10 sorted elements
- deallocate the array

We also use the `date_and_time` intrinsic subroutine to provide the timing details.

```
module sort_data_module
  implicit none

contains
  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
        end do
        do while (v<raw_data(j))
          j = j - 1
        end do
        if (i<=j) then
          t = raw_data(i)
          raw_data(i) = raw_data(j)
          raw_data(j) = t
          i = i + 1
          j = j - 1
        end if
        if (i>j) exit
      end do
      if (l<j) then
        call quicksort(l, j)
      end if
```

```
      if (i<r) then
        call quicksort(i, r)
      end if
    end subroutine quicksort
  end subroutine sort_data
end module sort_data_module

program ch2005
  use sort_data_module
  implicit none
  integer, parameter :: n = 10000000
  real, allocatable, dimension (:) :: x
  integer, dimension (8) :: timing
  real :: t1, t2
  character *30, dimension (4) :: heading = [ &
    ' Allocate =                 ', &
    ' Random number generation = ', &
    ' Sort =                     ', &
    ' Deallocate =               ' ]

  call date_and_time(values=timing)
  print *, ' Program starts'
  write (unit=*, fmt=100) timing(1:3), &
    timing(5:7)
100 format (2x, i4, 2('/',i2), ' ', 2(i2,':'), &
    i2)
  t1 = td()
  allocate (x(n))
  t2 = td()
  write (unit=*, fmt=110) heading(1), (t2-t1)
110 format (a30, f8.3)
  t1 = t2
!
! Random number generation
  call random_number(x)
  t2 = td()
  write (unit=*, fmt=110) heading(2), (t2-t1)
  t1 = t2
!
! Sorting
  call sort_data(x, n)
  t2 = td()
  write (unit=*, fmt=110) heading(3), (t2-t1)
  print *, ' First 10 sorted numbers are'
  write (unit=*, fmt=120) x(1:10)
```

```
120 format (2x, e14.6)
  t1 = t2
!
! Deallocation
  deallocate (x)
  t2 = td()
  write (unit=*, fmt=110) heading(4), (t2-t1)
  call date_and_time(values=timing)
  print *, ' Program terminates'
  write (unit=*, fmt=100) timing(1:3), &
    timing(5:7)

contains

  function td()
    real :: td

    call date_and_time(values=timing)
    td = 60*timing(6) + timing(7) + &
      real(timing(8))/1000.0
  end function td
end program ch2005
```

20.6.1 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.

20.6.2 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 routine.

A later chapter looks at a non recursive implementation of quicksort where we look at some of the ways of rewriting the above program by replacing the recursive quicksort with the non recursive version.

We call the `date_and_time` intrinsic subroutine to get timing information. The first three values are the year, month and day, and 5, 6 and 7 provide the hour minute and second. The last element of the array is milliseconds.

20.7 Example 6: Allocatable Dummy Arrays

In the examples so far allocation of arrays has taken place in the main program and the arrays have been passed into subroutines and functions.

In this example the allocation takes place in the `read_data` subroutine.

```
module read_data_module
  implicit none

contains
  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), allocatable, &
      dimension (:) :: raw_data
!   local variables
    integer :: i

    allocate (raw_data(1:how_many))
    open (unit=1, file=file_name, status='old')
    do i = 1, how_many
      read (unit=1, fmt=*) raw_data(i)
    end do
  end subroutine read_data
end module read_data_module

module sort_data_module
  implicit none
```

```
  contains
    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
          end do
          do while (v<raw_data(j))
            j = j - 1
          end do
          if (i<=j) then
            t = raw_data(i)
            raw_data(i) = raw_data(j)
            raw_data(j) = t
            i = i + 1
            j = j - 1
          end if
          if (i>j) exit
        end do
        if (l<j) then
          call quicksort(l, j)
        end if
        if (i<r) then
          call quicksort(i, r)
        end if
      end subroutine quicksort
    end subroutine sort_data
  end module sort_data_module
```

```
module print_data_module
  implicit none

contains
  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.txt', unit=2)
    do i = 1, how_many
      write (unit=2, fmt=*) raw_data(i)
    end do
    close (2)
  end subroutine print_data
end module print_data_module

program ch2006
  use read_data_module
  use sort_data_module
  use print_data_module
  implicit none
  integer :: how_many
  character (len=20) :: file_name
  real, allocatable, dimension (:) :: raw_data
  integer, dimension (8) :: timing

  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)
  call read_data(file_name, raw_data, how_many)
  call date_and_time(values=timing)
  print *, ' allocate and 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.txt'
end program ch2006
```

We now have a choice of where we do the allocation. This is more flexible than having to do the allocation in the main program, which is effectively a more Fortran 77 style of programming.

20.8 Example 7: Elemental Subroutines

We saw an example in Chap. 12 of elemental functions. Here is an example of an elemental subroutine.

```
module swap_module
  implicit none

contains
  elemental subroutine swap(x, y)
    integer, intent (inout) :: x, y
    integer :: temp

    temp = x
    x = y
    y = temp
  end subroutine swap
end module swap_module

program ch2007
  use swap_module
  implicit none
  integer, dimension (10) :: a, b
  integer :: i

  do i = 1, 10
    a(i) = i
    b(i) = i*i
  end do
  print *, a
  print *, b
  call swap(a, b)
```

```
  print *, a
  print *, b
end program ch2007
```

The subroutine is written as if the arguments are scalar, but works with arrays! User defined elemental procedures came in with Fortran 95.

20.9 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.

20.10 Problems

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

```
program ch2008
  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)
  end do
100 format (f8.3)
end program ch2008
```

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?

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

Was it worth writing a program?

20.3 Consider the following program:

```
program ch2009

! 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 ch2009
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 a module procedure (we haven't provided an interface for the subroutine) 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?

20.4 Use the intrinsic functions `matmul` and `transpose` to replace the current Fortran 77 style code in program ch2003.

20.11 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, ISBN 0-13-021999-1.

20.12 Commercial Numerical and Statistical Subroutine Libraries

There are two major suppliers of commercial numerical and statistical libraries:

- NAG: Numerical Algorithms Group

and

- Rogue Wave Software

They can be found at:

```
https://www.nag.co.uk/
```

and

```
https://www.roguewave.com/
```

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. In a later chapter we look at using a sorting routine from the Nag SMP & Multicore library.