Chapter 21
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.
- Modules containing procedures
- A module for timing programs
- Public, private and protected attributes
- The use statement and its extensions

21.1 Introduction

We have now covered the major executable building blocks in Fortran and they are

- The main program unit
- functions
- subroutines

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_21

and these provide us with the tools to solve many problems using just a main program, and one or more external and internal procedures. Both external and internal procedures communicate through their argument lists, whilst internal procedures have access to data in their host program units.

We have also introduced modules. The first set of examples was in the chapter on functions. The second set were in the chapter on derived types and the third set were in the subroutine chapters.

We will now look at examples of modules in more detail for

- Precision definition.
- Global data
- Modules containing procedures
- Derived type definition
- Simple timing information of a program

Modules provide the code organisational mechanism in Fortran and can be thought of as the equivalent of classes in C++, Java and C#. They are one of the most important features of modern Fortran.

21.2 Basic Module Syntax

The form of a module is

```
module module_name
..
..
..
end module module_name
```

and the specifications and definitions 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, function or subroutine statement.

21.3 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.

21.4 Example 1: Modules for Precision Specification and Constant Definition

In the chapter on arithmetic we introduced the features in Fortran that enable us to specify the precision of real numbers.

For the real numeric kind types, we used

- `sp` - single precision
- `dp` - double precision
- `qp` - quad precision

and here is the Fortran code segment from the program example.

```
integer, parameter :: &
  sp = selected_real_kind( 6, 37)
integer, parameter :: &
  dp = selected_real_kind(15, 307)
integer, parameter :: &
  qp = selected_real_kind(30, 291)
```

In this example we are going to package the above in a module, and then use the module to enable us to choose a working precision for the program and associated functions and subroutines. This module will be referred to in many examples in the book.

We will also have a second module with a set of physical and mathematical constants.

```
module precision_module
  implicit none
  integer, parameter :: sp = selected_real_kind( &
    6, 37)
  integer, parameter :: dp = selected_real_kind( &
    15, 307)
  integer, parameter :: qp = selected_real_kind( &
    30, 291)
end module precision_module
```

```
module maths_module

  use precision_module, wp => dp

  implicit none

  real (wp), parameter :: c = 299792458.0_wp
! units m s-1

  real (wp), parameter :: e = &
    2.71828182845904523536028747135266249 77_wp

  real (wp), parameter :: g = 9.812420_wp
! 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 (wp), parameter :: pi = &
    3.14159265358979323846264338327950288 4_wp

end module maths_module


module sub1_module
  implicit none

contains

  subroutine sub1(radius, area, circumference)

    use precision_module, wp => dp
    use maths_module
    implicit none
    real (wp), intent (in) :: radius
    real (wp), intent (out) :: area, &
      circumference

    area = pi*radius*radius
    circumference = 2.0_wp*pi*radius

  end subroutine sub1

end module sub1_module
```

```
program ch2101

  use precision_module, wp => dp
  use sub1_module

  implicit none

  real (wp) :: r, a, c

  print *, 'radius?'
  read *, r
  call sub1(r, a, c)
  print *, ' for radius = ', r
  print *, ' area = ', a
  print *, ' circumference = ', c

end program ch2101
```

In our example we have

```
use precision_module , wp => dp
```

and the `wp => dp` is called a `rename-list` in Fortran terminology. We are using it in this example to make `wp` point to the `dp` precision in the module.

Thus we can chose the working precision of our program very easily.

The kind type parameter `wp` is then used with all the real type declaration e.g.,

```
real (wp):: r ,a,c
```

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

```
2.0_wp
```

We set `e` and `pi` to over 33 digits as this is the number in a 128 bit real. This ensures that all calculations are carried out accurately to the maximum precision.

21.5 Example 2: Modules for Globally Sharing Data

The following example uses a module to define a parameter and two arrays. The module also contains three subroutines that have access to the data in the module. The main program has the statement

```
use data_module
```

which interfaces to the three subroutines.

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

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

contains
  subroutine readdata
    implicit none
    integer :: i
    character (len=40) :: filename

    print *, ' What is the filename ?'
    read *, filename
    open (unit=100, file=filename, status='old')
    do i = 1, n
      read (100, *) rainfall(i)
    end do
  end subroutine readdata

  subroutine sortdata
    implicit none

    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
    implicit none
    integer :: i

    print *, ' original data is '
    do i = 1, n
      print 100, rainfall(i)
    end do
    print *, ' Sorted data is '
    do i = 1, n
      print 100, sorted(i)
    end do
100 format (1x, f7.1)
  end subroutine printdata
end module data_module

program ch2102
  use data_module
  implicit none

  call readdata
  call sortdata
  call printdata
end program ch2102
```

21.6 Modules for Derived Data Types

When using derived data types and passing them as arguments to procedures, 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.

21.7 Example 3: 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_module` and then made the module available to each program unit with the statement

```
use personal_module
```

Note that we have put both subroutines in one module.

```
module personal_module
  implicit none
  type person
    real :: weight
    integer :: age
    character :: gender
  end type person
end module personal_module
module subs_module
  use personal_module
  implicit none
contains
  subroutine read_data(data, no)
    implicit none
    type (person), dimension (:), allocatable, &
      intent (out) :: data
    integer, intent (out) :: no
    integer :: i

    print *, 'input number of patients'
    read *, no
    allocate (data(1:no))

    do i = 1, no
      print *, 'for person ', i
      print *, 'weight ?'
      read *, data(i)%weight
      print *, 'age ?'
      read *, data(i)%age
      print *, 'gender ?'
      read *, data(i)%gender
    end do
  end subroutine read_data
```

```
  subroutine stats(data, no, m_a, f_a)
    implicit none
    type (person), dimension (:), &
      intent (in) :: data
    real, intent (out) :: m_a, f_a
    integer, intent (in) :: 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)%gender=='M' .or. &
                data(i)%gender=='m' &
        ) then
        m_a = m_a + data(i)%weight
        no_m = no_m + 1
      else if (data(i)%gender=='F' .or. &
          data(i)%gender=='f') then
        f_a = f_a + data(i)%weight
        no_f = no_f + 1
      end if
    end do
    if (no_m>0) then
      m_a = m_a/no_m
    end if
    if (no_f>0) then
      f_a = f_a/no_f
    end if
  end subroutine stats
end module subs_module
program ch2103
  use personal_module
  use subs_module
  implicit none
  type (person), dimension (:), allocatable :: &
    patient
  integer :: no_of_patients
  real :: male_average, female_average

  call read_data(patient, 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 ch2103
```

21.8 Example 4: A Module for Simple Timing of a Program

It is a common requirement to need timing details on how long parts of a program take. In this module we have a `start_timing` and `end_timing` subroutines and a `time_difference` real function. We will be using this module in several examples in subsequent chapters.

```
module timing_module

  implicit none
  integer, dimension (8), private :: dt
  real, private :: h, m, s, ms, tt
  real, private :: last_tt

contains

  subroutine start_timing()
    implicit none

    call date_and_time(values=dt)
    print 100, dt(1:3), dt(5:8)
    h = real(dt(5))
    m = real(dt(6))
    s = real(dt(7))
    ms = real(dt(8))
    last_tt = 60*(60*h+m) + s + ms/1000.0
100 format (1x, i4, '/', i2, '/', i2, 1x, i2, &
      ':', i2, ':', i2, 1x, i3)
  end subroutine start_timing

  subroutine end_timing()
    implicit none

    call date_and_time(values=dt)
    print 100, dt(1:3), dt(5:8)
100 format (1x, i4, '/', i2, '/', i2, 1x, i2, &
```

```
        ':', i2, ':', i2, 1x, i3)
    end subroutine end_timing

    real function time_difference()
      implicit none

      tt = 0.0
      call date_and_time(values=dt)
      h = real(dt(5))
      m = real(dt(6))
      s = real(dt(7))
      ms = real(dt(8))
      tt = 60*(60*h+m) + s + ms/1000.0
      time_difference = tt - last_tt
      last_tt = tt
    end function time_difference

end module timing_module
```

21.9 `private`, `public` and `protected` Attributes

With the examples of modules so far every entity in a module has been accessible to each program unit that 'uses' the module. By default all entities in a module have the public attribute, but sometimes it is desirable to limit the access. If entities have the private attribute this limits the possibility of inadvertent changes to a variable by another program unit.

Example of using public and private attributes:

```
real, public    : : a, b, c
integer, private :: i, j, k
```

If a variable in a module is declared to be public, its access can be partially restricted by also giving it the protected attribute. This means that the variable can still be seen by program units that use the module but its value cannot be changed e.g.

```
integer, public, protected:: i
```

21.10 The `use` Statement

In its simplest form the use statement is

```
use module_name
```

which then makes all the module's public entities available to the program unit. There may be times when only certain entities should be available to a particular program unit. In Example 1 subroutine `sub1` 'uses' `maths_module` but only needs `pi` and not `c`, `e` and `g`. The use statement could therefore be

```
use maths_module, only: pi
```

There are also times when an entity in a module needs to have its name changed when used in a program unit. For example variable `g` in `maths_module` needs to be called `gravity` in subroutine `sub1` so the use statement becomes

```
use maths_module, gravity=> g
```

We have also used this facility in example 1 where we renamed `dp` to `wp`.

21.11 Notes on Module Usage and Compilation

In the examples so far we have organised our code using one file. The file will comprise one or more of the following program units:

- main program
- subroutine
- function
- module

Another way of organising our code is to use several files and `include` statements.

The next example shows a way of doing this.

21.12 Example 5: Modules and Include Statements

Here is the program source.

```
include 'precision_module.f90'
include 'maths_module.f90'
include 'sub1_module.f90'
```

```
program ch2105

  use precision_module, wp => dp
  use sub1_module

  implicit none

  real (wp) :: r, a, c

  print *, 'radius?'
  read *, r
  call sub1(r, a, c)
  print *, ' for radius = ', r
  print *, ' area = ', a
  print *, ' circumference = ', c

end program ch2105
```

and we will use both styles throughout the rest of the book.

21.13 Formal Syntax

The following is taken from the Fortran standard and describes more fully requirements in the interface area.

21.13.1 Interface

The interface of a procedure determines the forms of reference through which it may be invoked. The procedures interface consists of its name, binding label, generic identifiers, characteristics, and the names of its dummy arguments. The characteristics and binding label of a procedure are fixed, but the remainder of the interface may differ in differing contexts, except that for a separate module procedure body (15.6.2.5), the dummy argument names and whether it has the NON_RECURSIVE attribute shall be the same as in its corresponding module procedure interface body (15.4.3.2).

An abstract interface is a set of procedure characteristics with the dummy argument names.

21.13.2 Implicit and Explicit Interfaces

Within the scope of a procedure identifier, the interface of the procedure is either explicit or implicit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit in such a scope.

The interface of a subroutine or a function with a separate result name is explicit within the subprogram where the name is accessible.

21.13.3 Explicit Interface

A procedure other than a statement function shall have an explicit interface if it is referenced and

- a reference to the procedure appears
  - with an argument keyword, or
  - in a context that requires it to be pure,
- the procedure has a dummy argument that
  - has the `allocatable`, `optional`, `pointer`, `target`, or `value` attribute,
  - is an assumed-shape array,
  - is a coarray,
  - is polymorphic,
- the procedure has a result that
  - is an array,
  - is a pointer or is allocatable, or
  - has a nonassumed type parameter value that is not a constant expression,
- the procedure is elemental

21.14 Summary

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

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

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

21.15 Problems

21.1 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.

21.2 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.