Chapter 27
Parameterised Derived Types (PDTs) in Fortran

Aims
The aims of this chapter are to look at some additional data structuring examples in Fortran that use parameterised derived types - PDTs.

27.1 Introduction

Parameterised derived types were introduced in the Fortran 2003 standard. They allow the kind, length, or shape of a derived type's components to be chosen when the derived type is used.

This feature was only available in two compilers (Cray and IBM) at the time of the second edition. Support for this feature is now available in three additional compilers. At the time of writing they were available in the following compilers:

- Cray
- IBM
- Intel
- Nag (partial)
- PGI

Consult our *Compiler Support for the Fortran 2003 and 2008 Standards* document

```
https://www.fortranplus.co.uk/
fortran-information/
```

for up to date information.

A parameterised derived type can have the kind, length and shape of a derived type chosen at run time. All type parameters are of type `integer` and have a `kind`, `len` or `dim` attribute. A kind type parameter may be used in constant and specification expressions. A length type parameter may only be used in a specification expression, e.g. array declarations.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_27

We have a small number of examples to illustrate their use.

27.2 Example 1: Linked List Parameterised by Real Kind

Here is the link module.

```
module link_module
  use precision_module
  type link(real_kind)
    integer, kind :: real_kind
    real (kind=real_kind) :: n
    type (link(real_kind)), pointer :: next
  end type link
end module link_module
```

Here is the complete program.

```
include 'precision_module.f90'
include 'ch2701_link_module.f90'

program ch2701
  use precision_module
  use link_module
  implicit none
  integer, parameter :: wp = dp
  type (link(real_kind=wp)), pointer :: root, &
    current
  integer :: i = 0
  integer :: error = 0
  integer :: io_stat_number = 0
  real (wp), allocatable, dimension (:) :: x

  allocate (root)
  print *, ' type in some numbers'
  read (unit=*, fmt=*, iostat=io_stat_number) &
    root%n
  if (io_stat_number>0) then
    error = error + 1
  else if (io_stat_number<0) then
    nullify (root%next)
  else
    i = i + 1
```

```
      allocate (root%next)
    end if
    current => root
    do while (associated(current%next))
      current => current%next
      read (unit=*, fmt=*, iostat=io_stat_number) &
        current%n
      if (io_stat_number>0) then
        error = error + 1
      else if (io_stat_number<0) then
        nullify (current%next)
      else
        i = i + 1
        allocate (current%next)
      end if
    end do
    print *, i, ' items read'
    print *, error, ' items in error'
    allocate (x(1:i))
    i = 1
    current => root
    do while (associated(current%next))
      x(i) = current%n
      i = i + 1
      print *, current%n
      current => current%next
    end do
    print *, x
  end program ch2701
```

Let us look at the `link_module` in more depth.

```
  type link(real_kind)
    integer, kind :: real_kind
    real (kind=real_kind) :: n
    type (link(real_kind)), pointer :: next
  end type link
```

The key is in the type declaration for `link` where the `link` type takes a parameter `real_kind`.

We then can reference this parameter within the `link` kind type definition. Thus the declarations for `n` and `next` are parameterised by `real_kind`.

In the main program we have

```
  integer, parameter :: wp = dp
  type (link(real_kind=wp)), pointer :: root, &
    current
```

and the type declarations for `root` and `current` are parameterised by `wp`, where `wp = dp`.

This means that we write one type definition for the `link` type that will work with any supported real kind type.

Without parameterised derived type support we would have to write separate kind type definitions for each supported real kind.

27.3 Example 2: Ragged Array Parameterised by Real Kind Type

Here is the ragged module.

```
module ragged_module
  use precision_module
  implicit none
  type ragged(real_kind)
    integer, kind :: real_kind
    real (real_kind), dimension (:), &
      allocatable :: ragged_row
  end type ragged
end module ragged_module
```

Here is the complete program.

```
include 'precision_module.f90'
include 'ch2702_ragged_module.f90'

program ch2702

  use precision_module
  use ragged_module
  implicit none
  integer, parameter :: wp = sp
  integer :: i
  integer, parameter :: n = 3

  type (ragged(wp)), dimension (1:n) :: &
    lower_diag
```

```
  do i = 1, n
    allocate (lower_diag(i)%ragged_row(1:i))
    print *, ' type in the values for row ', i
    read *, lower_diag(i)%ragged_row(1:i)
  end do
  do i = 1, n
    print *, lower_diag(i)%ragged_row(1:i)
  end do

end program ch2702
```

Let us look at the `ragged_module` in more depth.

```
module ragged_module
  use precision_module
  implicit none
  type ragged(real_kind)
    integer, kind :: real_kind
    real (real_kind), dimension (:), &
      allocatable :: ragged_row
  end type ragged
end module ragged_module
```

The key is in the type declaration for the `ragged` type.
We have

```
type ragged(real_kind)
```

so the kind definition is parameterised by `real_kind`.
The `ragged_row` array declaration is parameterised by `real_kind`.
In the main program we have

```
type (ragged(wp)), dimension (1:n) :: &
  lower_diag
```

so that the `lower_diag` declaration is parameterised by `wp`, where `wp = sp`.
So we have one declaration for the `ragged` type and can use this type with any supported real kind type.

27.4 Example 3: Specifying `len` in a PDT

In this example we use both the `kind` attribute and the `len` attribute in the type specification.

Here is the matrix module.

```
module pdt_matrix_module

  use precision_module
  implicit none

  type pdt_matrix(k, row, col)
    integer, kind :: k
    integer, len :: row, col
    real (kind=k), dimension (row, col) :: m
  end type pdt_matrix

  interface scale_matrix
    module procedure scale_matrix_sp
    module procedure scale_matrix_dp
  end interface scale_matrix

contains

  subroutine scale_matrix_sp(a, scale)
    type (pdt_matrix(sp,*,*)), intent (inout) :: &
      a
    real (sp) :: scale

    a%m = a%m + scale
  end subroutine scale_matrix_sp

  subroutine scale_matrix_dp(a, scale)
    type (pdt_matrix(dp,*,*)), intent (inout) :: &
      a
    real (dp) :: scale

    a%m = a%m + scale
  end subroutine scale_matrix_dp

end module pdt_matrix_module
```

Here is the complete program.

```
include 'precision_module.f90'
include 'ch2703_matrix_module.f90'

program ch2703
```

```
  use precision_module
  use pdt_matrix_module

  implicit none

  real (sp) :: scs
  real (dp) :: scd
  integer, parameter :: nr = 2, nc = 3
  integer :: i
  type (pdt_matrix(sp,nr,nc)) :: as
  type (pdt_matrix(dp,nr,nc)) :: ad
!
! single precision
!
  do i = 1, nr
    print *, 'input row ', i, ' of sp matrix'
    read *, as%m(i, 1:nc)
  end do
  print *, 'input sp scaling factor'
  read *, scs
  call scale_matrix(as, scs)
  print *, 'updated matrix:'
  do i = 1, nr
    print 100, as%m(i, 1:nc)
100 format (10(f6.2,2x))
  end do
!
! double precision
!
  do i = 1, nr
    print *, 'input row ', i, ' of dp matrix'
    read *, ad%m(i, 1:nc)
  end do
  print *, 'input dp scaling factor'
  read *, scd
  call scale_matrix(ad, scd)
  print *, 'updated matrix:'
  do i = 1, nr
    print 110, ad%m(i, 1:nc)
110 format (10(e12.5,2x))
  end do

end program ch2703
```

27.5 Problems

27.1 Modify example 1 to read the data from a file.

27.2 Rewrite the tree derived type in Chap. 22 as a parameterised derived type to work with an integer of any type. Test it out.