Chapter 17
Introduction to Derived Types

Russell's theory of types leads to certain complexities in the foundations of mathematics…Its interesting features for our purposes are that types are used to prevent certain erroneous expressions from being used in logical and mathematical formulae; and that a check against violation of type constraints can be made purely by scanning the text, without any knowledge of the value which a particular symbol might happen to have
C.A.R. Hoare, Structured Programming

Aims
The aim of this chapter is to introduce the concepts and ideas involved in using the facilities offered in modern Fortran for the construction and use of derived or user defined types;

- defining our own types.
- declaring variables to be of a user defined type.
- manipulating variables of our own types.
- nesting types within types.

The examples are simple and are designed to highlight the syntax. More complex and realistic examples of the use of user defined data types are to be found in later chapters.

17.1 Introduction

In the coverage so far we have used the intrinsic types provided by Fortran. The only data structuring technique available has been to construct arrays of these intrinsic types. Whilst this enables us to solve a reasonable variety of problems, it is inadequate for many purposes. In this chapter we look at the facilities offered by Fortran for the construction of our own types and how we manipulate data of these new, user defined types.

© Springer International Publishing AG, part of Springer Nature 2018

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_17

With the ability to define our own types we can now construct aggregate data types that have components of a variety of base types. These are given a variety of names including

- Record in the Pascal family of languages and in many older books on computing and data structuring;
- Structs in C;
- Classes in C++, Java, C# and Eiffel;
- Cartesian product is often used in mathematics and this is the terminology adopted by Hoare;

Chapter 3 has details of some books for further reading:

- Dahl O.J., Dijkstra E.W., Hoare C.A.R., Structured Programming;
- Wirth N., Algorithms + Data Structures = Programs;
- Wirth N., Algorithms + Data Structures.

We will use the term user defined type and derived types interchangeably.

There are two stages in the process of creating and using our own data types: we must first define the type, and then create variables of this type.

17.2 Example 1: Dates

```
program ch1701

  implicit none

  type date

    integer :: day = 1
    integer :: month = 1
    integer :: year = 2000

  end type date

  type (date) :: d

  print *, d%day, d%month, d%year
  print *, ' type in the date, day, month, year'
  read *, d%day, d%month, d%year
  print *, d%day, d%month, d%year

end program ch1701
```

This complete program illustrates both the definition and use of the type. It also shows how you can define initial values within the type definition.

17.3 Type Definition

The type `date` is defined to have three component parts, comprising a day, a month and a year, all of integer type. The syntax of a type construction comprises:

```
type typename
    data type :: component_name
    etc
end type typename
```

Reference can then be made to this new type by the use of a single word, `date`, and we have a very powerful example of the use of abstraction.

17.4 Variable Definition

This is done by

```
type (typename) :: variablename
```

and we then define a variable `d` to be of this new type. The next thing we do is have a `read *` statement that prompts the user to type in three integer values, and the data are then echoed straight back to the user. We use the notation

```
variablename%component_name
```

to refer to each component of the new data type.

17.4.1 Example 2: Variant of Example 1 Using Modules

The following is a variant on the above and achieves the same result with a small amount of additional syntax.

```
module date_module

  type date

    integer :: day = 1
    integer :: month = 1
    integer :: year = 2000

  end type date

end module date_module

program ch1702

  use date_module

  implicit none
  type (date) :: d

  print *, d%day, d%month, d%year
  print *, ' type in the date, day, month, year'
  read *, d%day, d%month, d%year
  print *, d%day, d%month, d%year

end program ch1702
```

The key here is that we have embedded the type declaration inside a module, and then used the module in the main program. Modules are covered in more detail in a later chapter.

If you are only using the type within one program unit then the first form is satisfactory, but if you are going to use the type in several program units the second is the required form.

We will use the second form in the examples that follow.

17.5 Example 3: Address Lists

```
module address_module

  type address

    character (len=40) :: name
    character (len=60) :: street
    character (len=60) :: district
```

```
    character (len=60) :: city
    character (len=8) :: post_code

  end type address

end module address_module

program ch1703

  use address_module

  implicit none

  integer :: n_of_address

  type (address), dimension (:), &
    allocatable :: addr

  integer :: i

  print *, 'input number of addresses'
  read *, n_of_address

  allocate (addr(1:n_of_address))

  open (unit=1, file='address.txt',status='old')

  do i = 1, n_of_address

    read (unit=1, fmt='(a40)') addr(i)%name
    read (unit=1, fmt='(a60)') addr(i)%street
    read (unit=1, fmt='(a60)') addr(i)%district
    read (unit=1, fmt='(a60)') addr(i)%city
    read (unit=1, fmt='(a8)') addr(i)%post_code

  end do

  do i = 1, n_of_address

    print *, addr(i)%name
    print *, addr(i)%street
    print *, addr(i)%district
    print *, addr(i)%city
    print *, addr(i)%post_code
```

```
    end do

end program ch1703
```

In this example we define a type `address` which has components that one would expect for a person's address. We then define an array `addr` of this type. Thus we are now creating arrays of our own user defined types. We index into the array in the way we would expect from our experience with integer, real and character arrays. The complete example is rather trivial in a sense in that the program merely reads from one file and prints the file out to the screen. However, it highlights many of the important ideas of the definition and use of user defined types.

17.6 Example 4: Nested User Defined Types

The following example builds on the two data types already introduced. Here we construct nested user defined data types based on them and construct a new data type containing them both plus additional information.

```
module personal_module

  type address

    character (len=60) :: street
    character (len=60) :: district
    character (len=60) :: city
    character (len=8) :: post_code

  end type address

  type date_of_birth

    integer :: day
    integer :: month
    integer :: year

  end type date_of_birth

  type personal
    character (len=20) :: first_name
    character (len=20) :: other_names
    character (len=40) :: surname
    type (date_of_birth) :: dob
    character (len=1) :: gender
```

```
    type (address) :: addr

  end type personal

end module personal_module

program ch1704

  use personal_module

  implicit none

  integer :: n_people
  integer :: i

  type (personal), dimension (:), &
    allocatable :: p

  print *, 'input number of people'
  read *, n_people

  allocate (p(1:n_people))

  open (unit=1, file='person.txt',status='old')

  do i = 1, n_people

    read (1, fmt=100) p(i)%first_name, &
      p(i)%other_names, p(i)%surname, &
      p(i)%dob%day, p(i)%dob%month, &
      p(i)%dob%year, p(i)%gender, p(i)%addr%street, &
      p(i)%addr%district, p(i)%addr%city, &
      p(i)%addr%post_code

  end do

  do i = 1, n_people

    write (*, fmt=110) p(i)%first_name, &
      p(i)%other_names, p(i)%surname, &
      p(i)%dob%day, p(i)%dob%month, &
      p(i)%dob%year, p(i)%gender, p(i)%addr%street, &
      p(i)%addr%district, p(i)%addr%city, &
      p(i)%addr%post_code
```

```
    end do

100 format (a20, /, a20, /, a40, /, i2, 1x, i2, &
    1x, i4, /, a1, /, a60, /, a60, /, a60, /, &
    a8)

110 format (a20, a20, a40, /, i2, 1x, i2, 1x, &
    i4, /, a1, /, a60, /, a60, /, a60, /, a8)

end program ch1704
```

Here we have a date of birth data type (`date_of_birth`) based on the `date` data type from the first example, plus a slightly modified `address` data type, incorporated into a new data type comprising personal details. Note the way in which we reference the component parts of this new, aggregate data type.

17.7 Problem

17.1 Modify the last example to include a more elegant printed name. The current example will pad with blanks the `first_ name`, `other_names` and `surname` and span 80 characters on one line, which looks rather ugly.

Add a new variable name which will comprise all three subcomponents and write out this new variable, instead of the three subcomponents.