Chapter 10

Structures and Derived Data Types

Modern programming languages provide programmers with the mean to create and manipulate new data type, much like the intrinsic data types, like integer, real, and logical encountered so far. In C and C++, these are called structures and in fortran they are called derived data type.

10.1 Derived Data Types

In many applications the intrinsic data types are not enough to express in code the ideas behind an algorithm or solution to a specific problem. Derived data types and structures allow programmer to group different kinds of information that belong to a single entity. In a way they resemble arrays but with two important differences. First the different elements of a derived data type do not have to be of the same type, thus they may include integer, character or real. Second the different entities making up the derived data type are referred to with a name and not an integer index. The different element of a derived data type are referred to as components. The data type of the component can be any of the intrinsic data types, or a previously defined derived data type.

10.2 Defining Derived Data Types

A derived data type is defined with a block of code enclosed within a type and end type statements followed by the name we wish to give to that derived data type. For example suppose we are writing a program that manipulates geometric information, and we would like to define points and polygons. Furthermore, we would like a point to have some attributes associated with it. For example if solving a PDE, we would like to know the type of boundary condition applied at that node.
A point data type in 2D space can thus be defined:

```fortran
type :: point
  real :: x(2) ! coordinates of point in 2D space
  integer :: bccode ! boundary condition code
end type point
```

The components of `point` are a real vector `x` of dimension 2, and an integer `bccode`. Once a type has been declared it can be used to declare variables of that type. For example the following code declares two variables, `A` and `B` and `C`, of type `point`:

```fortran
type(point) :: A,B,C
A = point( (/0.0,0.0/), 0) ! constructor for point A
B = point( (/1.0,0.0/), 0) ! constructor for point B
C = point( (/0.0,1.0/), 0) ! constructor for point C
```

The statement `A = point( (/0.0,0.0/), 0)` is a default constructor for data type `point`. The constructor is essentially like a function call whose name is that of the derived data type, and whose argument consist of the data type component listed in the order of their declaration in the data type definition; in this case it is the vector of real followed by the integer. One can also define arrays of derived data type, for example

```fortran
type(point) :: circle(64)
```

declares an 1D array of points of size 64.

A derived data type can be used as a component within another data type. For example, a polygon can now be defined as a collection of points, as in the following code:

```fortran
type :: polygon
  integer :: nnodes
  type(point), allocatable :: pointlist(:)
end type polygon
```

In this case the component consist of the number of points in the polygon and an allocatable array of `point` types.

### 10.3 Working with Derived Data Type

Each component of a derived data type can be addressed independently and used just like any other variable of the same type. If the component is an integer, it can be used like any other integer, and so forth. A component is specified with a component selector, which consists of the name of the variable followed by a percent sign `%` then by the component name. For example the following sets the coordinates of the variable `A`
10.4. Memory Allocation for Derived Data Type

When a variable of a derived data type is declared, the compiler has to allocate memory to all its components. The addresses of these components are not required to be in successive memory locations. Instead, the compiler is free to put the addresses in any location so it has enough leeway to optimize the memory access. However, sometimes a strict order of memory allocation is important (for example when passing the structure to a procedure written in a different programming language). If the element of a derived data type must be allocated in consecutive memory location for some reason, a `SEQUENCE` statement must be included in the derived data type definition. An example is

```fortran
type :: point
  sequence
  real :: x(2)
  integer :: bccode
end type polygon
```

10.5 Derived Data Type Declaration in Modules

The definition of a derived data type can be fairly bulky, and must be included in every procedure that uses variables of that type. To avoid the problem of duplicate declaration, it is most useful to define new data types in a module data section, and then to use the module in every procedure needing the data type. If the data type need modification, the changes to the definition can be performed in a single module and propagated throughout the code with the help of the `use` statement. Chapter 11.1 will present the use of modules for more effective programming in more details.
Chapter 11

Modules

11.1 Data Encapsulation

Modules are powerful tools that simplify the tasks of building complex large programs. Although we have encountered modules before we have not explored their full potential. This chapter explores some of the more advanced module constructs. We will use the example of derived data types to motivate and illustrate the use of these constructs.

Although it is easy to work with the components of a variable of a derived data type, it is not easy to work with the variables of derived data type as a whole. It is legal to assign one variable of given derived data type to another variable of the same type, essentially a copy operations of all the components, but other intrinsic operators, such as additions, substraction or multiplication are not defined. Operator overloading is one mechanism to define such operations on derived data types.

There is anoter reason where module use is important for derived data type: encapsulation. Encapsulation is the concept of packaging data and the methods that are used to manipulate that data is grouped together. The data and the methods would then be defined in the same module, and encapsulation requires that access to the data is done solely through the methods provided, i.e. through subroutine and function calls. In objected-oriented languages this packaging of data and methods is called a class. The encapsulation allows programmer to control access to the data types through interfaces. Changes to the data types, and the methods that manipulate them can be confined to one module that can be easily maintained, and are transparent to the procedure using those entities.

The points module shown in figure 11.1 is an example of a class which includes the data definition, the point derived type, and the methods that are used to operate on the class. These include querying and setting the various data components making up the derived type point. The contains statement separates the data declaration portion of a module from the procedures included in the module. In
module points
    implicit none
    integer, parameter :: sdim
    type :: point ! begin derived type definition
    real :: x(sdim) ! coordinate of points
    integer :: bccode ! some information about this node
    end type point ! end derived type definition
contains
!*******************************************************************************
subroutine GetCoordinate(pA, xc)
    type(point), intent(in) :: pA
    real, intent(out) :: xc(sdim)
    xc = pA%x ! retrieves the coordinates of a point
    return
end subroutine GetCoordinate
!*******************************************************************************
integer function GetBCCode(pA)
    type(point), intent(in) :: pA
    GetBCCode = pA%bccode ! retrieves the BC code of a point
    end integer GetBCCode
!*******************************************************************************
subroutine SetCoordinate(pA, xc)
    real, intent(in) :: xc(sdim)
    type(point), intent(inout) :: pA
    pA%x = xc ! sets the coordinates of a point
    end subroutine SetCoordinate
!*******************************************************************************
subroutine SetBCCode(pA, bccode)
    implicit none
    type(point), intent(inout) :: pA
    integer, intent(in) :: bccode
    pA%bccode = bccode ! sets the boundary condition code of a point
    end subroutine SetBCCode
!*******************************************************************************
subroutine PrintPoint(pA,unit)
    integer, intent(in) :: unit ! file unit to write to.
    type(point), intent(in) :: pA ! point to write
    write(unit,*) pA%x, pA%bccode
    return
end subroutine PrintPoint
!*******************************************************************************
subroutine ReadPoint(pA,unit)
    integer, intent(in) :: unit ! file unit to read from
    type(point), intent(in) :: pA ! point to write
    read(unit,*) pA%x, pA%bccode
    return
end subroutine ReadPoint
end module points

Figure 11.1: Point Module
program triangle
  use points
  implicit none
  type(point) :: tri(3)
call SetCoordinate(tri(1), (/0.0, 0.0/) ); call SetBCCode(tri(1), 1)
call SetCoordinate(tri(2), (/0.0, 1.0/) ); call SetBCCode(tri(1), 2)
call SetCoordinate(tri(3), (/1.0, 0.0/) ); call SetBCCode(tri(1), 3)
end program triangle

the present instance the data portion is used to declare the points data type. A main code wishing to use the methods included in the module would simply use the module. For example the following code declares an array of three points, to represent a triangle, iniatiates them and calculates the area of the triangle.

11.2 Private and Public Access

All variables and procedures defined in a module have their access set to public by default. That is procedures that use a module have access to all its data procedures. The initialization of the points in the triangle could have been done as follows:

tri(1)%x(1)=0.0; tri(1)%x(2)=0.0; tri(1)%bccode=1;

This statement, however, although permissible contradicts the principle of data hiding and encapsulation. The implementation of the data type is now visible to all part of the program, and changing (for whatever reason) later on would require changes to all parts of the code. In order to prevent other programmers to have access to the class implementation, it is useful to restrict access to the class' internal data. Access control to the data and procedure of a module is possible through the public and private statements. These statements can be used either as stand alone to declare the entire scope as either public or private, or they can be used as attributes to declare variables as public or private.

11.2.1 Access in type declaration

The following declaration restricts access of the members of a data type to the module where the data type is defined:

type :: point ! begin derived type definition
private ! no access granted to member variables
  real :: x(sdim) ! coordinate of points
  integer :: bccode ! some information about this node
end type point ! end derived type definition
Hence the program \texttt{triangle} must use \texttt{SetCoordinate} and \texttt{SetBCCCode} in order to initialize the points in \texttt{tri}(3). If the implementation of \texttt{points} changes later on, but the interface to its procedure kept fixed, then only changes are restricted to the file where \texttt{points} is defined; no changes to the other parts of the code are needed. It is important to note that in the example above only the member variables, \texttt{x} and \texttt{bccode}, are inaccessible, but the data type itself is available for use. Outside program units may freely declare variables of that data type, but may not work directly with its components. If the data type itself must be hidden then the following statement would be necessary:

\begin{verbatim}
type, private :: point  ! derived type definition is hidden within module
  real :: x(sdim)       ! coordinate of points
  integer :: bccode     ! some information about this node
end type point        ! end derived type definition
\end{verbatim}

11.2.2 Access for modules

By default all entities within a module have public access. To change it for all entities within a module it is enough to include the statement

\begin{verbatim}
module points
  private
  .
  .
end module points
\end{verbatim}

If only certain procedure or variable need to have their access changed then they must be listed as in:

\begin{verbatim}
module points
  private :: SetCoordinate, SetBCCCode
  .
  .
end module points
\end{verbatim}

The list can include a combination of variables and procedures declared within the module. The same construct applies to the \texttt{public} statement.

11.2.3 Access in variable declaration

The \texttt{points} define the dimension of the space within the module with the variable \texttt{sdim}. This variable may not be needed outside whereas all other entities defined in the module need be public. In this case it is better to leave the default access as is and only declare that single variable as private:
11.3 Scope of variables

In large programs variables may have duplicate names but refer to different entities. It is then important to have a mechanism to determine which definition of that name prevails in any one program unit. The concept of a **scope** is designed to manage the rules by which the ambiguity in the variable names is handled.

The scope of an object (a variable, name constant, procedure name or label name) is the portion of the program over which this object is defined. There are 3 levels of scope in Fortran:

1. **Global** Global objects are defined throughout an entire program. The names of these objects must then be unique. The only global objects encountered so far are the name of programs, external procedures and modules.

2. **Local** Local objects are defined and must be unique within a scoping unit. Examples of scoping units are main programs, external procedures and modules. A local object within a scoping must be unique within that unit, but the object name may be reused within another scoping unit without causing conflict.

3. **Statement** Local objects are defined and must be unique