Chapter 6: Object Oriented Programming

Object oriented programming (OOP) is a method that defines reusable structures which posess their own variables and functionality. These structures are then declared in specific instances when needed. Objects offer an effective way to operate on data using data types that are not inherent to Fortran. The basic property of Fortran that allows object-oriented programming is the ability to define new *data types*.

5.1 Derived Types

We have already learned that we are not limited to the intrinsic functions that Fortran offers; that is, you can create your own functions. The same is true of data types. You are not limited to the intrinsic data types such as `integer`, `real`, or `character`; you can create your own with the `type` declaration. For the following examples, new data types and functions that operate on them will be created modules, and these modules will be used by programs.

The following program illustrates how to declare, modify, and operate on a new data type called `point` that represents a 3D point.

type.f95

```
module pointmodule
implicit none
   ! define a new data type called point
   type point
      real :: x(3)
   end type point
contains
   ! define a function that operates on points
   function magnitude(pt)
      type(point), intent(in) :: pt
      real :: magnitude
      magnitude=sqrt(sum(pt%x**2))
   end function magnitude
end module pointmodule

program typeex
use pointmodule
implicit none
   ! declare a point
   type(point) :: pt

   ! set the values of 'x' in 'pt'
   pt%x(1)=1.
   pt%x(2)=2.
   pt%x(3)=3.
   ! print point and magnitude
   write(*,*) pt
   write(*,*) magnitude(pt)
end program typeex
```

- The new data type is created with the `type TYPE_NAME ...end type TYPE_NAME` tags. All variables that the new type contains are declared within these tags.
- After a new data type is created, an instance of it can be delcared with `type(TYPE_NAME) ::`

VAR_NAME where VAR_NAME is a named instance of the new data type. As a recognized data type, functions can be created whose arguments may be of the new data type.

- The variables of a derived data type can be accessed with the % symbol. For example, the variable x within an instance pt of the derived data type point is accessed by pt%x.

type - commands and output

```
gfortran type.f95 -o type
./type
   1.00000000        2.00000000        3.00000000
   3.74165750
```

The following program creates a matrix data type as well as some subroutines and functions that operate on the derived data type. The subroutines are designed to initialize a matrix as zeros, print a matrix, or free up the memory that a matrix occupies. The function is designed to compute the ∞-norm of a matrix.

matrix.f95

```fortran
module matrixmodule
implicit none
   ! define a new data type called matrix
   type matrix
      real, allocatable :: element(:,:)
   end type matrix
contains
   ! initialize matrix with zeros
   subroutine matrix_zeros(m, n, mat)
      integer, intent(in) :: m, n
      type(matrix), intent(out) :: mat
      integer :: i, j
      ! allocate memory for matrix
      allocate(mat%element(m,n))
      do i=1,m
         do j=1,n
            mat%element(i,j)=0.
         end do
      end do
   end subroutine matrix_zeros

   ! compute matrix inf-norm
   function matrix_infnorm(mat)
      type(matrix), intent(in) :: mat
      real :: matrix_infnorm
      integer :: m, i

      ! count number of rows
      m=size(mat%element(:,1))
      ! find maximum absolute row sum
      matrix_infnorm=0.
      do i=1,m
         matrix_infnorm=max(matrix_infnorm,sum(abs(mat%element(i,:))))
      end do
   end function matrix_infnorm

   ! print matrix
   subroutine matrix_print(mat)
      type(matrix) :: mat
      integer :: m, i

      ! count number of rows
```

```
      m=size(mat%element(:,1))
      ! print one row at a time
      do i=1,m
         print*, mat%element(i,:)
      end do
   end subroutine matrix_print

   ! destroy matrix
   subroutine matrix_destroy(mat)
      type(matrix) :: mat
      deallocate(mat%element)
   end subroutine matrix_destroy
end module matrixmodule

program matrixex
use matrixmodule
implicit none
   ! declare a matrix
   type(matrix) :: mat
   integer :: m=2, n=2, i, j

   ! initialize matrix as zeros
   call matrix_zeros(m,n,mat)
   write(*,*) '2x2 zero matrix: '
   call matrix_print(mat)
   ! assign elements of matrix
   do i=1,m
      do j=1,n
         mat%element(i,j)=real(i**j)
      end do
   end do
   write(*,*) 'new 2x2 matrix: '
   call matrix_print(mat)
   write(*,*) 'The inf-norm of the last matrix is ', matrix_infnorm(mat)
   ! destroy matrix
   call matrix_destroy(mat)

end program matrixex
```

matrix - commands and output

```
gfortran matrix.f95 -o matrix
./matrix
2x2 zero matrix:
  0.00000000       0.00000000
  0.00000000       0.00000000
new 2x2 matrix:
  1.00000000       1.00000000
  2.00000000       4.00000000
The inf-norm of the last matrix is    6.00000000
```

Exercise

1. Use the Thomas Algorithm (see Quateroni Section 3.7.1) to solve the system

$$\begin{bmatrix} 2 & \frac{1}{2} & & & & \\ \frac{1}{2} & 2 & \frac{1}{2} & & & \\ & \frac{1}{2} & 2 & \frac{1}{2} & & \\ & & \ddots & \ddots & \ddots & \\ & & & \frac{1}{2} & 2 & \frac{1}{2} \\ & & & & \frac{1}{2} & 2 \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{n-2} \\ s_{n-1} \\ s_n \end{bmatrix} = 6 \begin{bmatrix} 0 \\ f[x_0, x_1, x_2] \\ f[x_1, x_2, x_3] \\ \vdots \\ f[x_{n-3}, x_{n-2}, x_{n-1}] \\ f[x_{n-2}, x_{n-1}, x_n] \\ 0 \end{bmatrix}$$

   for $s = (s_0, \dots, s_n)^T$ where $n = 10$, $f(x) = 1/(1 + e^x)$ and $x_i = -1 + 2i/n$ for $i = 0, \dots, n$. Note $f[x_{i-1}, x_i, x_{i+1}]$ is the second-order Newton divided difference on nodes x_{i-1}, x_i, x_{i+1}. Report the s that you found as well as the number of computations and the number of storage locations used to compute s.