Chapter 6: Object Oriented Programming

Object Oriented Programming (OOP) is a programming method that represents concepts as "objects" that contain data and procedures which operate on the data. Objects offer an effective and structured way to organize computer programs.

In fortran, modules are one important component of (OOP). In this chapter we introduce the other component of OOP as well as expanding the power with which we use modules. Also, unlike with `C++`, fortran programmers have to handle their own constructors and destructors for objects. Generally, this means one needs to be careful that any memory allocation (in a constructor) has a matching memory deallocation (in a destructor). This is especially important when using dynamic array which we discuss in Chap. 7.

6.1 Derived Types

We have seen the available intrinsic data types, like INTEGER, but is this all that is available to us? It turns out no! We can create our own data types. Because these user defined types contain instances of the intrinsic data types they are called ***derived types.*** We'll give a few examples of derived types that could come up in different computing projects. We also show how to access elements of a derived type using a `%` command.

DerivedEx1.f90

```
PROGRAM DerivedEx1
   USE Constants
   TYPE Coordinate3D
      REAL(KIND=RP),DIMENSION(3) :: x ! stores (x,y,z)
   END TYPE Coordinate3D
   TYPE(Coordinate3D) :: p1

   p1%x(1) =  6.0_RP
   p1%x(2) =  0.0_RP
   p1%x(3) = -1.0_RP
   WRITE(*,*)p1
END PROGRAM DerivedEx1
```

DerivedEx2.f90

```
PROGRAM DerivedEx2
   USE Constants
   IMPLICIT NONE
   TYPE Coordinate3D
      REAL(KIND=RP),DIMENSION(3) :: x ! stores (x,y,z)
   END TYPE Coordinate3D
   TYPE Pixel
      TYPE(Coordinate3D) :: position
      INTEGER            :: color(3)
   END TYPE Pixel
!
   TYPE(Pixel) :: pixel1,pixel2

   pixel1%color(1) = 255
   pixel1%color(2) = 0
   pixel1%color(3) = 255
   pixel1%position%x(1) = 2.0_RP
   pixel1%position%x(2) = 200.0_RP
   pixel1%position%x(3) = 50.0_RP
   WRITE(*,*)pixel1
! Another way to set values in a derived type
   pixel2 = Pixel(Coordinate3D( (/ 15.0_RP/6.0_RP, 125.0_RP, 220.0_RP /) ),(/ 123,68,23 /))
```

```fortran
   WRITE(*,*)pixel2
END PROGRAM DerivedEx2
```

6.2 Example: Matrices

Now let's combine the concepts of derived types and modules to create a matrix object as well as functions and subroutines that operate on the matrix object. In the module which defines the matrix TYPE we use a standard notation of referring to the object to be operated on as `this`.

MatrixModule.f90

```fortran
MODULE MatrixModule
   USE Constants
   IMPLICIT NONE
   TYPE Matrix
      REAL(KIND=RP) :: element
   END TYPE Matrix
!
CONTAINS
!
   REAL(KIND=RP) FUNCTION OneNorm(this)
      TYPE(Matrix),INTENT(IN) :: this(:,:) ! means arbitrary size
! Local variables
      REAL(KIND=RP) :: colSum
      INTEGER       :: j,m,n

      m = SIZE(this(:,1))
      n = SIZE(this(1,:))
      OneNorm = 0.0_RP
      DO j = 1,n
         colSum  = SUM(ABS(this%(:,j)%element))
         OneNorm = MAX(OneNorm,colSum)
      END DO
   END FUNCTION OneNorm
!
   SUBROUTINE matrix_real_scalar(this,real_scalar)
      REAL(KIND=RP),INTENT(IN)  :: real_scalar
      TYPE(Matrix) ,INTENT(OUT) :: this

      this%element = real_scalar
   END SUBROUTINE matrix_real_scalar
!
   SUBROUTINE matrix_real_matrix(this,real_matrix)
      REAL(KIND=RP),INTENT(IN)  :: real_matrix(:,:)
      TYPE(Matrix) ,INTENT(OUT) :: this(:,:)

      this(:,:)%element = real_matrix(:,:)
   END SUBROUTINE matrix_real_scalar
END MODULE MatrixModule
```

MatrixMain.f90

```fortran
PROGRAM MatrixMain
   USE MatrixModule
   IMPLICIT NONE
   INTEGER,PARAMETER :: N = 2
   TYPE(Matrix)      :: mat1(N,N)
   INTEGER           :: i,j
   REAL(KIND=RP)     :: array(N,N)
```

```
   DO i = 1,N
      DO j = 1,N
         array(i,j) = 2.0_RP**i + 3.0_RP**j
      END DO
   END DO
   CALL matrix_real_matrix(mat1,array)
   WRITE(*,*)OneNorm(mat1)
   CALL matrix_real_scalar(mat1(1,2),2.3_RP)
END PROGRAM MatrixMain
```

These subroutines and function all work, and will yield correct values. However, it can become unwieldy to keep track of all the procedures in a module. We can simplify our lives if we use an INTERFACE.

6.3 Interfaces

We know the two major components of Object Oriented Programming in fortran. We know about derived types and can use modules to organize data, functions, and subroutines. Next let's learn about the INTERFACE construct, which will make modules easier to use.

MatrixModule2.f90

```
MODULE MatrixModule
   USE Constants
   IMPLICIT NONE
   TYPE Matrix
      REAL(KIND=RP) :: element
   END TYPE Matrix

   INTERFACE ASSIGNMENT(=)
      MODULE PROCEDURE matrix_real_scalar,matrix_real_matrix
   END INTERFACE
!
CONTAINS
!
   REAL(KIND=RP) FUNCTION OneNorm(this)
      TYPE(Matrix),INTENT(IN) :: this(:,:) ! means arbitrary size
! Local variables
      REAL(KIND=RP) :: colSum
      INTEGER       :: j,m,n

      m = SIZE(this(:,1))
      n = SIZE(this(1,:))
      OneNorm = 0.0_RP
      DO j = 1,n
         colSum  = SUM(ABS(this(:,j)%element))
         OneNorm = MAX(OneNorm,colSum)
      END DO
   END FUNCTION OneNorm
!
   SUBROUTINE matrix_real_scalar(this,real_scalar)
      REAL(KIND=RP),INTENT(IN)  :: real_scalar
      TYPE(Matrix) ,INTENT(OUT) :: this

      this%element = real_scalar
   END SUBROUTINE matrix_real_scalar
!
   SUBROUTINE matrix_real_matrix(this,real_matrix)
      REAL(KIND=RP),INTENT(IN)  :: real_matrix(:,:)
```

```
      TYPE(Matrix) ,INTENT(OUT) :: this(:,:)

      this(:,:)%element = real_matrix(:,:)
   END SUBROUTINE matrix_real_scalar
END MODULE MatrixModule
```

MatrixMain2.f90

```
PROGRAM MatrixMain2
   USE MatrixModule
   IMPLICIT NONE
   INTEGER,PARAMETER :: N = 2
   TYPE(Matrix)      :: mat1(N,N)
   INTEGER           :: i,j
   REAL(KIND=RP)     :: array(N,N)

   DO i = 1,N
      DO j = 1,N
         array(i,j) = 2.0_RP**i + 3.0_RP**j
      END DO
   END DO
!   CALL matrix_real_matrix(mat1,array)
   mat1 = array
   WRITE(*,*)OneNorm(mat1)
!   CALL matrix_real_scalar(mat1(1,2),2.3_RP)
   mat1(1,1) = 2.3_RP
END PROGRAM MatrixMain2
```

It doesn't look like we have done a lot, but the INTERFACE associated two subroutines in the module MatrixModule2 with the assignment = operator. (In effect we overloaded the equals operator to two different subroutines). So when we invoke the = operator the compiler will check to see if either of those subroutines work with the data types involved.

We can use an INTERFACE to simplify calls to fortran's intrinsic functions. For example, let's change a call to MATMUL into the * operator. We'll add this capability to the previous module.

MatrixModule3.f90

```
MODULE MatrixModule
   USE Constants
   IMPLICIT NONE
   TYPE Matrix
      REAL(KIND=RP) :: element
   END TYPE Matrix
   INTERFACE ASSIGNMENT(=)
      MODULE PROCEDURE matrix_real_scalar,matrix_real_matrix
   END INTERFACE
   INTERFACE OPERATOR(*)
      MODULE PROCEDURE matrix_matrix_multiply
   END INTERFACE
!
CONTAINS
!
   REAL(KIND=RP) FUNCTION OneNorm(this)
      TYPE(Matrix),INTENT(IN) :: this(:,:) ! means arbitrary size
! Local variables
      REAL(KIND=RP) :: colSum
      INTEGER       :: j,m,n

      m = SIZE(this(:,1))
```

```fortran
      n = SIZE(this(1,:))
      OneNorm = 0.0_RP
      DO j = 1,n
         colSum  = SUM(ABS(this%(:,j)%element))
         OneNorm = MAX(OneNorm,colSum)
      END DO
   END FUNCTION OneNorm
!
   SUBROUTINE matrix_real_scalar(this,real_scalar)
      REAL(KIND=RP),INTENT(IN)  :: real_scalar
      TYPE(Matrix) ,INTENT(OUT) :: this

      this%element = real_scalar
   END SUBROUTINE matrix_real_scalar
!
   SUBROUTINE matrix_real_matrix(this,real_matrix)
      REAL(KIND=RP),INTENT(IN)  :: real_matrix(:,:)
      TYPE(Matrix) ,INTENT(OUT) :: this(:,:)

      this(:,:)%element = real_matrix(:,:)
   END SUBROUTINE matrix_real_scalar
!
   FUNCTION matrix_matrix_multiply(this1,this2) RESULT(this_prod)
      TYPE(Matrix),INTENT(IN) :: this1(:,:),this2(:,:)
      TYPE(Matrix)            :: this_prod(SIZE(this1,1),SIZE(this2,2))!pull correct dimensions

      this_prod(:,:)%element = MATMUL(this1(:,:)%element,this2(:,:)%element)
   END FUNCTION matrix_matrix_multiply
END MODULE MatrixModule
```

MatrixMain3.f90

```fortran
PROGRAM MatrixMain3
   USE MatrixModule
   IMPLICIT NONE
   INTEGER,PARAMETER :: N = 3
   TYPE(Matrix)      :: mat1(N,N)
   INTEGER           :: i,j
   REAL(KIND=RP)     :: array(N,N)

   DO i = 1,N
      DO j = 1,N
         array(i,j) = 2.0_RP**i + 3.0_RP**j
      END DO
   END DO
!   CALL matrix_real_matrix(mat1,array)
   mat1 = array
   WRITE(*,*)OneNorm(mat1)
!   CALL matrix_real_scalar(mat1(1,2),2.3_RP)
   mat1(1,1) = 2.3_RP
   mat1 = mat1*mat1
   DO i = 1,N
      PRINT*,mat1(i,:)
   END DO
END PROGRAM MatrixMain3
```

You could also add subroutines to this module, and possibly INTERFACE functionality, for an inversion call A/B, an LU decomposition, the Thomas Algorithm, printing arrays, or other matrix operations.