# Computer Science 1510

Lecture 16

## Lecture Outline

- Derived data types
- Modules

# Derived data types

- A program combines an algorithm and data to solve a problem.

- So far we have seen how to use Fortran primitive data types such as INTEGER, REAL, and CHARACTER.

- We have also seen that arrays can be used to store lists of data of the same type.

- Depending on the data used for a particular problem, it may be useful to be able to group data using a more sophisticated group organization.

- A *derived data type* is a term used to refer to the grouping and organization of primitive data types into complex structures.

# Derived data types

- For example, suppose that we would like to have a data type called `Student` that would store information for a single student, including the following:
  - Student number
  - Last name
  - First name
  - Assignment grades
  - Lab quiz grades
  - Midterm grade
  - Final exam grade
  - Final grade

- This would require a much more complex data type than we have already seen. However, each component in this new data type is of an intrinsic type (ex. characters, integers).

# Derived data types

- In general, a derived data type is defined as follows:

```
TYPE type-name
   declaration of field names
END TYPE type-name
```

- We can define a derived type with the above components as follows:

```
TYPE Student
   CHARACTER(Len=9)::id_num
   CHARACTER(Len=15)::first_name
   CHARACTER(Len=20)::last_name
   REAL::assign(8),labs(10),mid,final,grade
END TYPE Student
```

- We now have a derived data type that can be used similar to an intrinsic data type.

# Derived data types

- We can declare variables to be of type `Student`. For example,

  ```
  TYPE(Student) :: st1
  ```

  where `st1` is called a *structure*.

- Each element in a data structure is called a *component*, or a *field*. For example, `id_num`, `first_name`, and `labs` are all fields in the `Student` data structure.

- To access an element of the data structure we use the % symbol.

  For example, to set the first name in the structure `st1` we would write:

  ```
  st1%first_name='John'
  ```

# Arrays of derived data types

- It is possible, indeed quite useful, to have arrays composed of *structures*, or derived data types.

- For example, if we have a class of 50 students then we can declare an array of type `Student` as follows:

```
TYPE(Student) :: Class(50)
```

- To access an element of the data structure we use the % symbol. For example, to set the first name of student 1 we would write:

```
Class(1)%first_name='John'
```

- `Class(1)` accesses the first element in the `Class` array, which is of type `Student`.

- `first_name` is a field within the student `Class(1)`.

# Example: TYPE

```
PROGRAM Structure
   IMPLICIT NONE
   ! Define a Student data structure
   TYPE Student
      CHARACTER(Len=9)::id_num
      CHARACTER(Len=15)::first_name
      CHARACTER(Len=20)::last_name
      REAL::assign(8),mid,final,grade
   END TYPE Student
   INTEGER::i,j,num,OpenStatus,InputStatus
   REAL::total,aavg
   CHARACTER(Len=20)::FileName

   ! Declare an array of 10 Students
   TYPE(Student)::Class(10)

   WRITE(*,'(A)',ADVANCE='NO') 'Which file should be read?  '
   READ(*,*) FileName
   OPEN(UNIT=5,FILE=FileName,STATUS='OLD',IOSTAT=OpenStatus)
   IF (OpenStatus > 0) STOP

   ! Determine how many students are in the class
   READ(5,*,IOSTAT=InputStatus) num
   IF (InputStatus /= 0) STOP

   ! Read in the student data
   DO i=1,num
      READ(5,*,IOSTAT=InputStatus) Class(i)%id_num
      IF (InputStatus /= 0) STOP
      READ(5,*,IOSTAT=InputStatus) Class(i)%first_name
      IF (InputStatus /= 0) STOP
      READ(5,*,IOSTAT=InputStatus) Class(i)%last_name
      IF (InputStatus /= 0) STOP
      READ(5,*,IOSTAT=InputStatus) Class(i)%assign
      IF (InputStatus /= 0) STOP
```

```
      READ(5,*,IOSTAT=InputStatus) Class(i)%mid
      IF (InputStatus /= 0) STOP
      READ(5,*,IOSTAT=InputStatus) Class(i)%final
      IF (InputStatus /= 0) STOP
   END DO

   ! For each Student, compute the final mark
   DO j=1,num
      ! Compute the average assignment mark for Student j
      total=0
      DO i=1,8
         total=total+Class(J)%assign(i)
      END DO
      aavg=total/8.0

      ! Compute the final mark for Student j
      ! Assignments marked out of 20, midterm and final marked out of 50.
      ! Assignments = 30%, Midterm = 30%, Final = 40%
      Class(j)%grade=aavg/20.0*30.0+Class(j)%mid/50.0*30.0+Class(j)%final/50.0*40.0

      ! Display the final mark for Student j
      WRITE(*,10) Class(j)%first_name,Class(j)%last_name,'received',&
                  Class(j)%grade,'in the course.'
   END DO
10 FORMAT(A15,A20,X,A8,X,F5.1,X,A14)

   CLOSE(UNIT=5)
END PROGRAM Structure
```

# Output:

```
John           Doe                  received  74.2 in the course.
Jane           Smith                received  71.3 in the course.
```

# Example input file

```
2
123456789 - student ID
John - first name
Doe - last name
19,17,15,10,16,17,18,14 - assignments
35 - midterm
37 - final
987654321 - student ID
Jane - first name
Smith - last name
16,18,14,13,10,11,18,16 - assignments
32 - midterm
38 - final
```

# Fortran MODULE

- A MODULE is a program unit used to package together related information and functions.

- Variable declarations, subprograms, and definitions of new data types are items that are commonly found in a `MODULE`.

- Syntax:

```
MODULE module-name
   IMPLICIT NONE
   Variable-declarations
CONTAINS
   Subprogram-definitions
END MODULE module-name
```

- A MODULE can be placed before the main program in a .f08 file or in its own file (see later).

- Modules allow related data and/or subprograms to be grouped together to be used by different programs as needed.

# Example 1: MODULE

```
MODULE Circle
!-----------------------------------------------------------------
! The following module contains subprograms to compute attributes
! of a circle:
!    Area: compute the area of a circle
!    Circumference: compute the circumference of a circle
!-----------------------------------------------------------------
   IMPLICIT NONE
   REAL,PARAMETER::PI=3.14159

CONTAINS
   !-Area-----------------------------------------------------------
   ! Compute the area of a circle with radius r.
   !  Accepts: radius of the circle
   !  Returns: the area of the circle
   !-----------------------------------------------------------------
   FUNCTION Area(r)
      REAL,INTENT(IN)::r
      REAL::Area
      Area=PI*r*r
      RETURN
   END FUNCTION Area

   !-Circumference--------------------------------------------------
   ! Compute the circumference of a circle with radius r.
   !  Accepts: radius of the circle
   !  Returns: the circumference of the circle
   !-----------------------------------------------------------------
   FUNCTION Circumference(r)
      REAL,INTENT(IN)::r
      REAL::Circumference
      Circumference=2*PI*r
      RETURN
   END FUNCTION Circumference
END MODULE Circle
```

# Using a MODULE

- In order to have access to the contents of a MODULE, a program (or subprogram) must request access via a USE statement.

- Syntax:

```
USE module_name
```

- A USE statement must occur after the program or subprogram name and before the declaration statements.

- Modules allow restricted access to subprograms since only programs and subprograms that USE the MODULE are permitted to call subprograms within the MODULE.

# Example 1: Using a MODULE

```
PROGRAM Calc_Circle
!-----------------------------------------------------------------
! The following program calculates and displays the area and
! circumference of a circle of a given radius.
!  INPUT:  r - radius of the circle
!  OUTPUT: Area and circumference of the circle
!-----------------------------------------------------------------
   USE Circle
   IMPLICIT NONE
   REAL::r
   WRITE(*,*) 'Enter circle radius'
   READ(*,*) r
   WRITE(*,*) 'Circle area = ',Area(r)
   WRITE(*,*) 'Circle circumference = ',Circumference(r)
END PROGRAM Calc_Circle
```