Lecture 4b – Advanced Fortran 95/03

- **There are different data types used in Fortran 95/03**
  - Some compilers will delineate REAL variables into KINDS
  - KINDS are described in Chapman
  - Most of the time, you do not have to worry about the KINDS but will instead decide in advance if you wish to work in *single* or *double precision* and then declare variables accordingly
  - REAL*4 (single precision), REAL*8 (double precision)

- **Most modern scientific Fortran programs use *derived data type variables* to better describe their meaning**
  - Convenient way to group together all information regarding a particular item and their relationships
  - Used extensively in object-oriented programming (OOP)

Derived Data Types

- **Usage:**

```
TYPE :: type_name
component definitions
…
END TYPE type_name
```

- **EXAMPLE:**

```
TYPE :: PERSON
   CHARACTER (LEN=14) :: first_name
   CHARACTER :: middle_initial
   CHARACTER (LEN=14) :: last_name
   CHARACTER (LEN=14) :: phone
   INTEGER :: age
   CHARACTER :: sex
   CHARACTER (LEN=11) :: ssn
END TYPE PERSON
```


Derived Data Types

- **We could initialize variables or arrays as data types**

```
EXAMPLE:   TYPE (PERSON) :: JOHN, JANE
           TYPE (PERSON) :: DIMENSION(100) PEOPLE
           ....
           JOHN = PERSON('John','R','Jones','323-6439','21','M','123-45-6789')
           JANE  = PERSON('Jane','C','Doe','332-3060','19','F',999-99-9999')
           PEOPLE(1) = PERSON('Sam','L','Van','666-2354','33','M','987-65-4321')
```

- **We can refer to individual parameters of a derived data type (such as first_name, age, or ssn in the above example) by simply referring to type%parameter**

```
EXAMPLE:   JOHN%age = 21
           JANE%last_name = 'Doe'
```

- **You can now begin to see how all of the parameters (or characteristics) of a variable can be linked together by using derived data types**
  - Working with the parameters of derived data types is easy but working with the data types as a whole is not!

Derived Data Types in Modules

- **Derived data types can be defined in Modules and then "used" in different routines**
  - Input into Modules is the same as previously described

- **You can see how derived data types might be very useful in**
  - Data bases
  - Data structures used for scientific simulations
    - Unstructured-grids for instance:

```
TYPE GRID_CELL
   REAL :: cell_number
   CHARACTER(LEN=4) :: cell_type
   REAL :: number_of_faces
END TYPE GRID
```

  - Object oriented programming (OOP)

Dynamic Memory Allocation

- **One reason that the C computer language became so popular was its ability to dimension arrays dynamically.**
  - This eliminates the need to re-compile codes in order to handle different problem sizes
- **Fortran did not allow for dynamic memory allocation until the 90/95 versions (several years after C was able to)**
- **An array can be declared with variable dimensions through use of the ALLOCATABLE statement**

  REAL, ALLOCATABLE, DIMENSION (:,:) :: ARRAY1

  This declares ARRAY1 with a variable dimension to be defined with an ALLOCATE statement

ALLOCATE and DEALLOCATE

- **Once an array is declared ALLOCATABLE, the dimensions can be defined and the array created**

```
EXAMPLE: INTEGER, PARAMETER :: IDIM=200
         INTEGER, PARAMETER :: JDIM=100
         REAL, ALLOCATABLE DIMENSION (:,:) :: ARRAY1
         …
         ALLOCATE(ARRAY1(IDIM,JDIM),STAT=STATUS)
         …use in computation
         DEALLOCATE(ARRAY1,STAT=STATUS)
```

Where STATUS = 0 for successful allocation and a positive number if the allocation fails

Checking Allocated Arrays

- **Arrays can not be re-allocated once they have been allocated. They must first be deallocated and the allocated with new parameters.**
- **A way to check to see if an array has be allocated is to use the logical intrinsic function *ALLOCATED***

```
EXAMPLE: REAL, ALLOCATEABLE, DIMENSION (:) :: INPUT
         …
         IF(ALLOCATED(INPUT)) THEN
           READ(8,*) INPUT
         ELSE
           WRITE(*,*) 'Warning: INPUT not allocated'
         END IF
```


Pointers

- **Pointers are primarily used when variables and arrays are created and destroyed dynamically and we do not know in advance the number of each that will be needed**
- **A pointer is a variable that contains the address in memory of another variable where data is stored**
  - Pointer → address of variable
  - Variable → data value

Pointers

- **A Fortran variable is declared to be a pointer by including the POINTER attribute in its declaration statement**

  EXAMPLE: REAL, POINTER :: P1

- **A pointer is only allowed to point to variables of its declared type**
- **Pointers may also be used for derived data types**

  EXAMPLE: TYPE(VECTOR), POINTER :: VECTOR_P1
- **Pointers may also point to arrays**

  EXAMPLE: INTEGER, DIMENSION(:), POINTER :: P1
  REAL, DIMENSION(:,:), POINTER :: P2

Targets

- **A pointer can point to any variable or array of the pointer's type as long as the variable has been declared to be a *target***
- **A *target* is a data object whose address has been made available for use with pointers**
- **A Fortran variable or array may be declared to be a target by including the TARGET attribute in its declaration**

  EXAMPLE: REAL, TARGET :: A1 = 7
  INTEGER, DIMENSION(10), TARGET :: INT_ARRAY

Pointers → Targets

- **Pointers may be associated with a given target by using a *pointer assignment statement***

  EXAMPLE: pointer => target

  This stores the memory address of target in the pointer
- **We can also assign the value of one pointer to another pointer by using the assignment statement**

  EXAMPLE: pointer2 => pointer
- **Even though the address is stored in a pointer, reference to the pointer (say with a read or write statement) will result in the variable value that it is pointing to**

Disassociation and Status of Pointers

- **Pointers can be disassociated from their targets by using the *nullify* statement**

  EXAMPLE: NULLIFY(pointer1, pointer2,…)
- **The status of the association of a pointer can be determined using the *associated* statement**

  EXAMPLE: status = ASSOCIATED(pointer1)
  - where status is TRUE if the pointer is associated with any target

  or status = ASSOCIATED(pointer1,target1)
  - where status is TRUE if the pointer is associated with the particular target included in the ASSOCIATED statement

Pointers and Arrays

- **Pointers can also point to a targeted array. The pointer must declare the type and the rank of the array that it will point to:**

```
EXAMPLE:    REAL, DIMENSION(50,50), TARGET :: MYDATA
            REAL, DIMENSION(:,:), POINTER :: POINTER1
            POINTER1 => MYDATA
```

- **Pointers can also point to a subset of an array**

```
EXAMPLE:    REAL, DIMENSION(50,50), TARGET :: MYDATA
            REAL, DIMENSION(:,:), POINTER :: POINTER1
            POINTER1 => MYDATA(1:20,10:50)
```


Dynamic Memory Allocation of Array Pointers

- **A powerful feature of pointers is that they can be used to dynamically create variables or arrays (similar to what we did before but without the need for ALLOCATABLE statements)**

```
EXAMPLE:
   PROGRAM MEM
   IMPLICIT NONE
   INTEGER :: I, ISTAT
   INTEGER, DIMENSION(:) POINTER :: PTR1,PTR2
   ALLOCATE(PTR1(1:10),STAT=ISTAT)
   ALLOCATE(PTR2(1:10),STAT=ISTAT)
!  NOTE THAT WE DID NOT HAVE TO DECLARE PTR1 OR PTR2
!  AS ALLOCATABLE ARRAYS
   PTR1 = (/ I,I=1,10) /)
   PTR2 = (/ I,I=11,20) /)
```


Pointers with Derived Data Types

- **Another powerful use of pointers is with components of derived data types.**
  - Pointers in derived data types may even point to the derived data type being defined (circular!)
  - This feature allows us to construct types of dynamic data structures linked together by successive pointers during program execution. Such a structure is called a ***linked list***
  - A linked list is a series of variables of a derived data type with the pointer from each variable pointing to the next variable in the list

```
EXAMPLE:    TYPE :: REAL_VALUE
               REAL :: VALUE
               TYPE(REAL_VALUE), POINTER :: PNTR
            END TYPE
```

Contains a real number and a pointer to another variable of the same type

Uses of Link Lists

- **Link lists are useful when the size of the data set is dynamic and it is unknown at the start of the program what the final size will be**
  - Grid embedding adaptation is an example
  - Convergence histories is another example
- **Link lists allow us to add elements to an array, one at a time**

Pointers in Procedures

- **Pointers can also be used as dummy arguments in procedures or passed as arguments to procedures**
  - If a procedure has dummy arguments with either the POINTER or TARGET attributes, then the procedure must have an explicit interface
  - If a dummy argument is a pointer, then the actual argument passed to the procedure must be a pointer of the same type, kind, and rank
  - A pointer dummy argument cannot have an INTENT attribute
- **A function result can also be a pointer.**

Object Oriented Programming (OOP)

- **Modern programming methods include object oriented styles that include:**
  - Modular program design (intensive use of subroutines)
  - Use of re-usable programming "objects"
  - Abstract data types
  - General data structures
  - Pointers and targets
- **OOP methods have been used in the C language for several years and are now fully-functional in Fortran**
  - Object-based modular structure
  - Data abstraction
  - Automatic memory management
  - Classes
  - Inheritance
  - Polymorphism (generic functionality) and dynamic binding

Data Types

- **We've mentioned some data types used in programming**
  - Intrinsic types: character, logical, floating point, complex, real
  - User defined data types: derived data types
- **Now let's discuss data types and some definitions in the context of OOP**
- **Abstract Data Types**
  - Coupling or encapsulation of the data with a select group of functions defining everything that can be done with the data → abstract data type
  - Abstraction (generalization):
    - pertains to only the essential features of the data
    - Features are defined in a manner that is independent of any specific programming language
    - Instances are defined by their behavior and the implementation is secondary

Classes

- **Class: an extension of an abstract data type by providing additional member routines to serve as constructors**
- **The constructor ensures that the class is created with acceptable values assigned to all its data attributes.**
  - Example a type:

```
type (face_information) :: face
  integer :: face_index
  integer :: face_idim, face_jdim
  real*8, allocatable, dimension(:,:) :: xface,yface
end type face_information
```

when combined with a constructor to create the information contained in the type and all included in a module, it becomes a class

Object, Instance, Encapsulation, and Data Hiding

- **Object: combines various classical data types into a set that defines a new variable type or structure.**
- **Instance: every object from a class, by providing the necessary data, is called an instance of the class.**
- **Encapsulation: the coupling or encapsulation of data and its functions into a unified entity. This is performed by including the class in a module.**
- **Data hiding: the protection of information in one part of a program from access and from being changed in other parts of the program. In the module, the "contains" statement couples the data, specifications, and operators before it to the functions and subroutines that follow it in the module.**

Inheritance and Polymorphism

- **Inheritance: we can employ one or more previously defined classes (of data and functionality) to organize additional classes. Functionality programmed into the earlier classes may not need to be recoded to be usable in the later class.**
  - This is performed in Fortran with "use" followed by the name of the module statement block that defined the class.
- **Polymorphism: ability of a function to respond differently when supplied with arguments that are objects of different types**

Example

- **In this class (and likely during your careers) you will develop (or use) a differential equation solver for modeling some physical phenomena.**
  - Heat conduction, stress/strain, flow
- **In such solvers, there are several kernels that make up the algorithms and several algorithms that make up the code**
  - Grid cell face area
  - Grid cell volume
  - i-face, j-face, k-face fluxes
  - Integration of fluxes
  - Timestep size
  - Update of variables, etc.

Procedural vs OOP

- **Most people learn how to program codes start off learning the procedural programming style**
- **Let's say you wanted to start programming your project-1 code**
- **The pseudo-code for a procedural program to solve the heat conduction equation might look something like the following:**

Procedural Programming Example-1

```
program heat
!  this program solves the 2D heat conduction equation on a structured grid
   create global computational grid
      define computational and physical dimensions
      allocate memory for grid and temperature variables
   initialize temperature on computational grid
   set the boundary conditions on temperature
   solve the heat conduction equation
      calculate constant-i face primary projected lengths
      calculate constant-j face primary projected lengths
      calculate primary cell volume
      calculate primary cell-centered time-step size
      calculate constant-i face temperature fluxes
      calculate constant-j face temperature fluxes
      calculate primary cell center temperature first-derivatives
```


Procedural Programming Example-1

```
      calculate constant-i face secondary projected lengths
      calculate constant-j face secondary projected lengths
      calculate secondary cell volume
      calculate secondary node-centered time-step size
      calculate constant-i face temperature derivative fluxes
      calculate constant-j face temperature derivative fluxes
      calculate secondary node-center temperature second-derivatives
      calculate right-hand side of heat conduction equation
         sum second derivatives and multiply by time-step size
            these are the changes in temperature
      apply boundary conditions
      update temperature and test for convergence
      if (notconverged) iterate
      else write out final solution in standard (plot3d) format
   end program
```


Procedural Programming Example-1

- **Each line of the pseudo-code could be either programmed directly in the main program (that I do not recommend), or written as a subroutine that is called from the main program.**
- **When subroutines are used, you need to get information exchanged between the subroutine and the main program**
  - This can be done using arguments in the subroutine call OR a module that can be “used” in both the subroutine and the main program
  - Also note that subroutines that pertain to the calculation of geometry variables do not need to be placed inside of the iteration loop

Procedural Programming Example-2

```
program heat
use variable_module
!  this program solves the 2D heat conduction equation on a structured grid
call computational grid creation
   define computational and physical dimensions
   allocate memory for grid and temperature variables and store in
      variable_module
call temperature initialization
call boundary condition initialization
call constant-i face primary projected lengths
call constant-j face primary projected lengths
calculate primary cell volume
calculate constant-i face secondary projected lengths
calculate constant-j face secondary projected lengths
calculate secondary cell volume
```

Procedural Programming Example-2

```
solve the heat conduction equation
  call primary cell-centered time-step size
  call constant-i face temperature fluxes
  call constant-j face temperature fluxes
  call primary cell center temperature first-derivatives
  call secondary node-centered time-step size
  call constant-i face temperature derivative fluxes
  call constant-j face temperature derivative fluxes
  call secondary node-center temperature second-derivatives
  call right-hand side of heat conduction equation
    sum second derivatives and multiply by time-step size
      these are the changes in temperature
  call boundary conditions
  call update temperature and test for convergence
  if (notconverged) iterate
  else write out final solution in standard (plot3d) format
end program
```


Procedural Programming Example-2

- **Then the subroutines are linked to the main program during the linking step, after compilation of all subroutines, in the creation of the code**
- **The subroutines can be included in the same file as the main program or in separate files**

- **Procedural programming is the most straightforward method for engineers. However, it lacks the generality (abstractions) and organization that OOP offers.**

OOP Example-3

```
  program heat
!  this program solves the 2D heat conduction equation on a structured grid
  use grid_module
  use face_module
  use cellvolume_module
  use facefluxes_module
  use secondaryface_module
  use secondarycellvolume_module
  use secondaryfacefluxes_module
  use boundarycondition_module
  create global computational grid (class)
    define computational and physical dimensions
    allocate memory for grid and temperature variables and store in module
  call temperature initialization (subroutine)
  call boundary condition initialization (subroutine)
  call constant-i face primary projected lengths (class)
  call constant-j face primary projected lengths (class)
```


OOP Example-3

```
calculate primary cell volume (class)
calculate constant-i face secondary projected lengths (class)
calculate constant-j face secondary projected lengths (class)
calculate secondary cell volume (class)
solve the heat conduction equation
  call primary cell-centered time-step size
  call constant-i face temperature fluxes (class)
  call constant-j face temperature fluxes (class)
  call primary cell center temperature first-derivatives (class)
  call secondary node-centered time-step size (class)
  call constant-i face temperature derivative fluxes (class)
  call constant-j face temperature derivative fluxes (class)
  call secondary node-center temperature second-derivatives (class)
```

OOP Example-3

```
call right-hand side of heat conduction equation (class)
   sum second derivatives and multiply by time-step size
      these are the changes in temperature
 call boundary conditions (class)
 call update temperature and test for convergence
 if (notconverged) iterate
 else write out final solution in standard (plot3d) format
end program
```


OOP Example-3

- Each class consists of a module of the allocatable array(s) pertaining to that particular class and contains the subroutine that creates the data. For instance, constant-i face lengths class would be:

```
module constant-i_face_lengths
real*8, allocatable, dimension(:,:) :: dx_i, dy_i
contains
use dimensions_module (contains imax, jmax)
do j = 1,jmax-1
  do i = 1,imax
    dx_i(i,j) = x(i,j+1)-x(i,j)
    dy_i(i,j) = y(i,j+1)-y(i,j)
  enddo
enddo
end module constant-i_face_lengths
```


Projects

- **During the creation of your projects, I would like for you to start by using procedural programming until you are comfortable with programming techniques.**
- **However, I would like for you to move toward OOP as soon as possible and incorporate abstract programming techniques with classes to make your codes more elegant and general.**

Homework 2 (cont)

- **Read Chapters 11-15 in Fortran95/03 (Chapman)**