• In our driver program to call the appropriate nonlinear solver, we had to make extensive use of conditionals. In particular, we used the `select case` construct.

• We had to hardwire the call to each nonlinear solver and we had to code different statements to get ready for the next iteration as well as to read in starting values.

• Consequently, we had to modify our driver program in several places when we added another nonlinear solver.

• Wouldn’t it be nice if we could just have statements like
call read_in_initial_values()
xkp1 = next_iterate()
call get_ready_for_next_iteration()

and the code would call the appropriate routine based upon the arguments used.

• Object-oriented programming allows us to do this.
• Objected-Oriented Programming (OOP for short) is becoming increasingly popular.

• High level objected-oriented (OO) languages provide the user with three main capabilities that improve and simplify the design of codes.
  – encapsulation
  – inheritance
  – polymorphism (generic functionality)

• Related ideas include objects, classes and data hiding.

• An OO language provides a way to couple or encapsulate the data and its functions into a unified entity. Fortran 90 uses modules for encapsulation which we have briefly introduced.

• Inheritance in OOP has a similar meaning to the common usage of the word. Suppose we have defined a class in a module and then we want to write a second class which uses the first. Fortran allows us to inherit functionality from the first class into the second; i.e., we don’t have to reprogram it in the second unit. A nontechnical example of this would be if we defined a class of objects called employees (with pertinent information like SSN, DOB,
etc) and then defined a second class called managers. Since managers are also employees we would like to “inherit” all the functionality of the employee class and then add the additional information for managers that is not needed for regular employees.

• Polymorphism allows different classes of objects that share some common functionality to be used in code that requires only that common functionality. In other words, routines having the same generic name are interpreted differently depending on the class of objects presented as arguments to the routines. Some intrinsic functions have always had this capability. For example, real (a) will convert an integer to a real if a is an integer whereas if a is complex, it give the real part. Fortran 90 allows generic functions to also have this feature.

• We will look at several examples of data types and how they can be used.
- The main problem we are going to address in this section of the course is computing with vectors. This will require us to learn about one-dimensional arrays in fortran. We want to write a vector class with associated subprograms and incorporate many of the concepts of OOP.
A Simple Example to Illustrate Polymorphism and Inheritance

• In the next problem we are going to look at, we will be using an OO approach to writing our code. To understand some of the OO concepts, we first look at some simple problems.

• Our first goal is to write an OO code which does the following.
  – We will define two derived data types (i.e., data structures) - a rectangle and a circle. The definition of the data structure along with the supporting subprograms is called a class.
  – We will have two function routines in modules - one to calculate the area of a circle if its radius is given and another to calculate the area of a rectangle if its two sides are given.
  – We will have a main or driver program which constructs either a rectangle or a circle and then computes the area using a single calling statement like

\[
\text{area} = \text{compute\_area}(\text{argument}).
\]
If the argument is an object in the rectangle class then it calls the routine for calculating the area of a rectangle; if the argument is an object in the circle class it calls the routine for calculating the area of a circle. Remember that the two routines for calculating area must have different names.

- If we can do this, then it will be an example of **polymorphism**. That is, the program behaves differently (it calls different function routines) depending on what “type” the argument is. If we can do this, then it will be our first OO code. This is also called **function overloading**.

- **What new fortran commands do we need to accomplish our goal of coding this example of polymorphism?**
  - **Interface** statement - will allow us to call functions with different names by one generic name (function overloading); e.g., `compute_area` could be the generic statement and it will call either `compute_area_circle` or `compute_area_rectangle` depending on the type (i.e., what class it belongs to) of the argument.

  - **Type** statement - will allow us to define a **data structure or derived data**
type; this definition, along with supporting subprograms will form our rectangle or circle class. One of these objects will be the argument for compute_area.

• We will write a module containing our two function routines for calculating area, say compute_area_rectangle and compute_area_circle. We have already written modules. For example, we put all of our subprograms for finding a nonlinear equation into a module.

• Now we will view a module from a different viewpoint - as a means for defining a class of objects and encapsulating all the routines related to that class in one unit, i.e., the module.
Interface Statement for Function Overloading

- The `interface` statement will allow us to give a generic name to several routines. (It has other uses which we will see later.)

- We must provide
  - the generic name, and
  - a list of all routines that can be called using this generic name.

- Where do we put the `interface` statement?
  - Probably the best place to put it is in the module before the `contains` although it can be put in the main program.
  - This means that in our module for the rectangle we will have an interface statement for the routine `compute_area_rectangle` and another statement in the module for the circle for the routine `compute_area_circle`. 
Syntax for Interface Statement for Function Overloading

**interface** name
  module procedure list of subprogram names
end interface

For our example we could have for the module *class_rectangle*

**interface compute_area**

  module procedure compute_area_rectangle

end interface
• The main program uses both modules and each module procedure will be added to a list of procedures that have the generic name `compute_area`. Everytime you invoke `compute_area` in the main program, the compiler checks the argument to see if the argument is an object in the circle class or in the rectangle class that you have defined (we will see how to do this in a minute) and calls the appropriate subprogram.

• Of course the main program must know about the classes that we have defined in our modules.

• For the main program to have this functionality, we must bring in the information from our modules. As we did before, this is done by simply invoking the command `use module name` at the first of our main program. So the main program is inheriting the functionality (i.e., the classes, etc.) that we defined in our modules.

• If we list a function name in the module procedure list which doesn’t exist, we will get an error. If the subprogram exists in the module, but we fail to tell the main program about it (i.e., we didn’t include the `use module` statement) then we will get an error.
Derived Data Types/Classes and the TYPE Construct

- Allowing the user to define a class as a data structure is a powerful tool. The terminology class or data type or derived data type is used synonymously.

- We will see that to store elements of all the same type (integer or real, etc.) we can use an array. This can be a one-dimensional array (i.e., a vector), a two-dimensional array (i.e., a matrix) or a higher dimensional array.

- To store elements of different types (or the same type) which are related in some way, we can use a derived data type for a data structure defined through the type construct.

- For a derived data type we want to name the data type (or data structure) and give the attributes (or components) which define an object in the data type.

- For our simple example, we know that we can find the area of a circle if we know the radius and the area of a rectangle if we know the length and the width.
• Consequently, when we define the data structure for a circle we would only need to specify one component, the radius.

• For a rectangle we need two components, the length and width.
Fortran 90 uses the `type` statement to define a derived data type; this along with associated subprograms will form a class.

**Syntax for Type Statement**

```
  type name
      list of components with declaration
  end type name
```
A sample rectangle data structure whose components are real variables width and height.

```fortran
type rectangle
  real(prec) :: width
  real(prec) :: height
end type rectangle
```

A sample circle data structure whose only attribute is the real variable radius.

```fortran
type circle
  real(prec) :: radius
end type circle
```
• What do we need to define a data structure?
  – The **name** of the data type.
  – Each attribute or component of the data structure (such as length and width of rectangle).
  – Declaration of type (integer, real, etc) of each attribute.

• If we define a data structure called element (for a periodic element) we might include its symbol (a character), its atomic number (an integer), its atomic weight (a real number), etc.

• A useful consequence of using a data structure which has several declared data types is that instead of passing all the variables through arguments of a calling statement we can simply pass the name of the object in the class. This will reduce errors concerning mismatch of arguments and makes the code more readable.

• There is a mechanism to access a particular component of an object in a data type; this can be done via the `%` symbol.
Where do we put our data structure definitions prescribed via the `type` statement?

- We view a module as encapsulating all routines for a particular class of objects.
- Consequently we should define the derived data type at
  - the `beginning of the module`
  - `after the implicit none statement` (because you will be declaring variables)
  - `before contains`
The Module Revisited

The structure of a module with a data type definition

module name

implicit none

type statements

interface statements

contains

subprograms (functions and subroutines) related to defined class

end module name
How does the main program know that an object is a member of a data structure?

- Recall that we include a `use module name` in the main program to inherit the data type definitions from the module.

- Suppose we have defined a data structure called `rectangle` with attributes `width` and `height`.

- To identify an object as a member of this data type then we must declare it as such, just like we declare a variable as a real, integer, etc.

- The syntax we use is, e.g.,

  ```
  type(rectangle):: rect
  ```

  This defines the object `rect` as a member of the data structure `rectangle`.

- We do not have to declare its attributes because this is done in the `type` statement in the module.

- When we pass `rect` as an argument to a subprogram, then we are really passing the two variables `width` and `height`. 
• How can we access one of the attributes of a member of a data structure? For example, if we pass \texttt{rect} how do we compute the area?

  – The syntax \texttt{rect \% width} says to look at the object \texttt{rect} which has been defined by the \texttt{type} statement as a member of the data structure \texttt{rectangle} and access the attribute \texttt{width}. If we typed \texttt{print *, rect\%width} then the current value of the width would be printed. Using this notation, the area of a rectangle could be computed as

    \[
    \text{area} = \texttt{rect \% width} \times \texttt{rect \% length}
    \]

  – Note that we have to use the \texttt{exact name} that we declared as an attribute in the \texttt{type} statement. That is, we couldn’t say \texttt{rect \% w} because \texttt{w} is not an attribute of a member of the rectangle data structure. This would generate a compiler error.

  – If we type \texttt{print *, rect} then it will print out the components of this member of the rectangle data structure in the order which they were defined (here \texttt{width}, then \texttt{length}).
Sample module for defining a rectangle class

module class_rectangle

use common_data

implicit none

type rectangle ! definition of rectangle data structure

    real(prec) :: width, height

end type rectangle

interface compute

    module procedure compute_area_rectangle

end interface

end module
contains

function compute_area_rectangle ( rect ) result(area)

type (rectangle) :: rect

real(prec) :: area

area = rect % width * rect % height

end function compute_area_rectangle

end module class_rectangle

Note here that we have chosen to use the result specifier for the function so that the output of the function will be returned in the variable area instead of the function name. This is simply another way to code this. Note that in this case the name of the function doesn’t have to be declared since nothing is returned in it; however we must declare the variable area since it is output.
• We first define the derived data type `rectangle` with two attributes which we declare.

• We can call the function `compute_area_rectangle` by using its name.

• However, because we included the `interface` construct we can also access it with the call `compute_area`.

• If we want to invoke `compute_area_rectangle` using `compute_area` then it only works when the argument is a member of the rectangle data structure. Our `interface` construct is tied to the `type` definition.

• In the function routine we had to define the object `rect` as a member of the rectangle derived data type.
Sample module for defining a circle class

module class_circle

use common_data
implicit none

type circle ! defines circle data structure
    real(prec) :: radius
end type circle

interface compute_area

    module procedure compute_area_circle

end interface

end module class_circle
contains

function compute_area_circle ( circ ) result(area)

type (circle) :: circ

real(prec) :: area

area = pi * (circ % radius)**2

end function compute_area_circle

end module class_circle

• Note that for this module we need \( \pi \) but this is defined through our common_data module.

• We can put both of these modules into a single file, say class_geometry.f90 or in separate files. For now, we will put them in a single file until we learn to use a “make file”. Remember that the name of the file goes on the gfortran line, not the name of the module.
Other terminology which is used in OOP is that of constructors and destructors for objects in a data structure.

We may write our own constructor for a class or use an intrinsic constructor.

For example, to define an object rect in a rectangle data structure we could write

```plaintext
rect % width = 2.0; rect % length = 4.0
```

where we have defined a rectangle with width 2.0 and length 4.0.

Alternately, we can use the intrinsic constructor

```plaintext
rect = rectangle ( 2.0, 4.0 )
```

Note that the order is important; the value 2.0 is assigned to the first attribute defined in the type rectangle. For us, this sets the width to 2.0 and the height to 4.0 since our definition of the rectangle data structure was
type rectangle  
    real :: width, height  
end type rectangle  

• Here rect had to be previously defined as a member of the rectangle data structure through the declaration

  type (rectangle) :: rect  

• It is possible to use the intrinsic constructor when we are setting all of the attributes of an object in a data structure. However, it will often be the case that we do not have all of this information and will have to write our own constructor. We will see an example of this in a minute.

• The destructor is used to reinitialize the data structure definition or release storage when it is no longer needed.
A Sample Main or Driver Program

- We need to include the `use module` statements.
- We will use our intrinsic constructors to construct a rectangle and a circle.
- We will use a call to compute the area using the generic name `compute_area`. 
program geometry

use class_circle; use class_rectangle; use common_data

implicit none

type (rectangle) :: rect ! declare rect as member of rectangle data structure

type (circle) :: circ ! declare circ as member of circle data structure

real(prec) :: area

rect = rectangle (two, four) ! use intrinsic constructor to define rect

circ = circ (two) ! use intrinsic constructor to define circ

area = compute_area (rect) ! argument is object in rectangle data structure so calls compute_area_rectangle
write(*,1001) area

area = compute_area(circ) ! argument is object in circle data structure so calls compute_area_circle

write(*,1002) circ % radius, area

1001 format(" area of rectangle is ", f12.5)

1002 format("area of circle with radius", f12.5, "is", f12.5)

end program geometry
Summary

- In the module
  - We define our data structure or derived type; declare all of its attributes as real, integer, etc.
  - Add an interface statement for a routine that is called via a generic call; you must give the generic call as well as the name of the specific routine to be called this way.
  - The first argument in the subprogram should be a member of the defined data structure.
  - In the subprograms the variable which is a derived type must be declared that via a type declaration.

- In the main calling program
  - We must add use module name. This gives the main program access to the subprograms as well as informing it of our derived type and its attributes.

  use class_rectangle; use class_circle
– We must declare each member of a derived type as such through `type` declaration.

– When we use the generic call to a subprogram we must make sure the first argument in the list is a member of the desired derived data type.

• In the `gfortran` line we include the `.o` files that contain the modules needed.

  `gfortran geometry.f90 class_geometry.o`
As an example of a manual constructor, let's consider a slightly modified problem. Suppose we want to have the option of creating a square where we just need to input say the width and not the length. We could either define a new derived data type `square`, we could use the `rectangle` type and input the same width and length or we can use another aspect of fortran90.

Fortran allows us the capability of having an optional argument. Assume the length is declared optional. If both the width and length are passed to the constructor, then it constructs a rectangle using

\[ \text{rect} = \text{rectangle} \left( \text{width, length} \right) \]

but if only the width is passed,

\[ \text{rect} = \text{rectangle} \left( \text{width} \right) \]

then it creates the rectangle as if we had passed

\[ \text{rect} = \text{rectangle} \left( \text{width, width} \right) \]
• Two new aspects of Fortran will be used. The *optional* specifier on a declaration statement; for example

```fortran
real(prec), optional :: length
```

and the *present* logical intrinsic function.

• The conditional

```fortran
if ( present( length ) ) then
```

returns *true* if an actual argument was passed for the optional argument `length`.

• An example of such a manual constructor is given and it is included as a function in the module `class_rectangle`. 
function make_rectangle ( width, length ) result(rect)
imPLICIT none
type (rectangle) :: rect
real :: width
real, optional :: length

if ( present(length ) ) then ! construct rectangle
   rect = rectangle ( width, length)
else ! default to square
   rect = rectangle ( width, width )
end function make_rectangle

• Note that the length has an optional specifier which means it does not have to be present in the argument statement.

• The use of present(argument) is a logical function that tests whether the argument is passed to the subprogram.

• As we do more complicated problems we will see that a constructor and deconstructor are integral parts of our modules.
How would we modify the manual constructor for the rectangle class so that it does all of the following?

- If neither the width nor the length is present, set the rectangle to the default value of a one by one square.
- If the length is not present, set the rectangle to a square of side given by the width.
- If both length and width are present set the rectangle to the prescribed width and length.
Defining Other Geometric Classes

• Clearly we could define data structures for other geometric shapes (triangles, pentagons, etc.) or we could define a more general polygon class.

• However, we want to treat this as a simple example of our first OOP code and move on to applying these ideas to problems in scientific computing.

• For homework you will be asked to add subprograms to compute the perimeter of your objects using a generic name `compute_perimeter`, to add a right triangle class, and manual constructors.

• REMEMBER: if you modify the module file you have to recompile it.
Classwork

Trying out the code

- Compile the modules `class_rectangle` and `class_circle` in the file `class_geometry.f90` Notice that two `mod` files are created and only one object file `class_geometry.o`.

- Compile the driver program `geometry.f90` remembering on the command line that you have to give it access to the object file `class_geometry.o` (which contains our two modules).

- Execute the program. It should calculate the area of a circle of radius two.

- Use the intrinsic constructor to set the values of a rectangle to say width=3.5 and length = 2.0; add a statement to compute the area of a rectangle.

- Add a `write` or `print` statement for the rectangle so that it prints out the width, length and resulting area of the rectangle. Use the “%” symbol to do
this. Run your code and make sure it is working.

- Try out the manual constructor using just the width so that it computes the area of a square. Test it.

- Modify the manual constructor for the rectangle class so that it does the following:
  - If neither the width nor the length is present, set the rectangle to the default value of a one by one square.
  - If the length is not present, set the rectangle to a square of side given by the width.
  - If both length and width are present set the rectangle to the prescribed width and length.
Homework

- HwkIII_1