Goals of this Lecture

• Because defining classes (i.e., derived data types or data structures) are an important aspect of OOP, we want to look at additional examples to illustrate some other concepts.

• Recall that classes are a type of data structure which allows us to store objects of different types.

• We will introduce the OOP concept of operator overload which is similar to function overload.

• In this lecture and the next we will look at a class for rational numbers and we will introduce the concepts of public and private.

• Next week we will begin to look at one-dimensional arrays or vectors which allow us to store elements of the same type.
An Example of a Data Structure
one of whose Components is itself a Data Structure

• When we define a data structure with the type construct we sometimes want to extend these attributes to define another data structure which has the attributes of the first plus some additional components.

• A simple, non-scientific example of this is defining an employee data structure.

• Suppose we want the attributes of the employee data type to contain
  – first and last name (characters)
  – id number (an integer)
  – date of birth (month, day and year)
  – date started working (month, day and year)
  – current hourly wage (floating point)

Note how these components are of different types.
• Notice that two of the items are dates. So it makes sense to first define a date data structure.

• Each date should contain the month (an integer from 1 to 12), a day of the month (an integer from 1 to 31), and a year (a 4 digit integer). So we might want to define a class/derived data type called `date` which has 3 integer components (month, day, year). Then each of the third and fourth components of our employee class is itself a class and we will declare it as such.

• Examples of these two data structures are given below.
module class_employee
use common_data
implicit none

type (date)
  integer :: month, day, year
end type date

type (employee)
  character ( len = 30) :: first_name
  character ( len = 30) :: last_name
  integer :: id
  type(date) :: dos
  type(date) :: dob
  real(prec) :: hourly_wage
end type employee
contains
  :
end module class_employee
• Note that the way we declared the date of birth (dob) to be a member of the class date is the same syntax that we declared rect to be a member of the rectangle class in our main program geometry.f90.

• We could define separate modules for these two classes or we could put them in the same module. For ease of use I have put them into the same module called class_employee.

• We want to add a subprogram to check that the date is valid and then print out the date in a format such as February 14, 2010. Should it be a function or subroutine?

• What would the conditional look like to check if the month and date is valid?
Suppose that the subroutine statement is

```fortran
subroutine print_date(mdy)
```

where `mdy` is a member of the `date` class.

Remember that when we pass `mdy` we are really passing three integer values, the month, the day and the year although we only have one argument. This is because `mdy` is a data structure (derived type) and must be declared as such with the statement

```fortran
type (date) :: mdy ! define mdy as object in date class
```

To check to see if we have input a valid month and day our code could look like the following
if ( mdy % month < 1 .or. mdy % month > 12 ) then ! invalid month
    print *, " invalid month"
else if ( mdy % day < 1 .or. mdy % day > 12 ) then! invalid day
    print *, " invalid day"
else ! print out date
    statements
end if
• I have also added a routine for printing out the employee’s information, that is, all the information in the class employee called `print_employee_info`.

• To print out the first and last name on a line, I have used the `trim` intrinsic function. Here `trim(string)` removes all trailing blanks from `string`.

• For example,

  ```fortran
  print *, " Employee name: ", trim( joe % first_name), " ", trim(joe % last_name)
  ```

  produces

  Employee name: John Doe

  while without the `trim` we get something like

  Employee name: John Doe

• This intrinsic is sometimes useful to make your output look nice when you use an unformatted write.
Now we want to write a main program which tests out this employee class.

1. First, we will declare an object `today` which is a member of the data structure `date`; set its value to today’s date using the intrinsic constructor and print out.

```fortran
program test_employee_class

use class_employee

implicit none

! declare today a member of the data structure date

type(date) :: today

! intrinsic constructor for date

today = date ( 2, 11, 2010 )
call subroutine print_date ( today )
```
end program test_employee_class

Output:

February 11, 2010
2. Define an object in the employee class and set its components using the intrinsic constructor. Use Jane Doe, a starting date of January 7, 2010, a six digit integer id, a date of birth and hourly wage. Print out this employee information.

- We need to add a declaration for the sample employee to be a member of the employee data structure.

```fortran
type (employee) :: sample
```

- Remember that the attributes of the employee data structure are (in order): `first_name, last_name, id, dos, dob, hourly_wage`

- Because two `(dos, dob)` are members of the `date` structure we define them first using the intrinsic constructor.

```fortran
dos = date(1, 7, 2010)
dob = date(7, 3, 1989)
```

- We now use the intrinsic constructor for the data structure `employee` to set all values.
sample = employee ( 'Jane', 'Doe', 124561, dos, dob, 12.50 )
call print_employee_info (sample)

Output:

Employee name: Jane Doe
Employee ID number: 124561
Employee Starting Date: January 7, 2010
Employee Date of Birth: July 3, 1989
3. Now we want to modify our codes so that in the main program we can call either `print_date` or `print_employee_info` using the generic name `print_info`. This is an application of function overloading which is a form of polymorphism.

How can we do this?
• In order to be able to call either routine by the generic name `print_info` we must add an `interface` construct in the module.

```plaintext
interface print_info       ! the generic calling name
    module procedure print_date, print_employee_info
end interface
```

• Note that we listed both routines in one interface construct. Alternately, I could have had two interface constructs with the same name `print_info` but just list one routine in each.

• Now in the main program we can use the following statements:

```plaintext
call print_info (dos)
```

To print out a date because `dos` has been declared a member of the date data structure.

```plaintext
call print_info (sample)
```

To print out employee information because `sample` has been declared a member of the employee data structure.
4. Your job for classwork will be to define a data structure called `manager` which has attributes:

- all employee information (don’t enter separately just declare that data structure)
- annual salary (real number)
- whether manager is assistant manager or manager (character)
- date promoted to manager status

You will write a routine to print out a manager’s information and add statements to call it with the generic name `print_info`. 
An Example of a Numerical Calculation Class - Rational Numbers

- Recall that a **rational number** can be represented as the ratio of two integers.

- In this example we will define a data structure for a rational number and add some subprograms related to performing operations on rational numbers (such as adding them, converting a rational number to a real number, etc.)

- We will introduce the concept of **public versus private**.

- We will look at another aspect of OOP called **operator overloading**.
• Because a rational number is determined by the integer numerator and integer denominator, we will define our class (data structure) for a rational number as

```plaintext
type rational
    integer :: num
    integer :: den
end type rational
```

• Our goal is to add functionality to our module so that we can add or subtract two rational numbers, multiply two rational numbers, reduce a rational number to its simplest form, print out a rational number, convert it to a floating point number, etc.

• The derived data type plus these subprograms and any interface statements will form our rational number class.
Public and Private Accessibility

• By default all module variables and subprograms are available to all program units which have the use module name statement.

• This may not always be desirable. If the module procedures (given in the interface construct) provide all the access that is necessary for users, it may be safer to not allow users the ability to interfere with the internal workings of your module.

• By default, all names in a module are public but this can be changed with the private statement.

• Since everything is public by default, you may choose to just declare the variables or routines that you want private; for example

```module class_rational_numbers
implicit none
private :: gcd, reduce```
Here everything is public except the subprograms `reduce` and `gcd` which are routines that are only called by other public subprograms in the module. The user would not even know of their existence unless he/she looked at the actual code.

- If we have declared `reduce` as `private` and we have a call in the main program to this subroutine we will get the compile time error message

  ```
  undefined symbols : reduce
  ```

  just as if the program did not exist.

- Alternately, you could declare everything as private and just list what you want to be public. For example,

  ```
  module class_rational_numbers
  implicit none
  private
  public :: add_rational
  ```

Here the entire routine is private and only the subprogram
add_rational is public.

- For now we will mainly use the private specifier if our routine is only to be used by other subprograms in the module.
Operator Overloading

- Suppose we write a subprogram to add two members of our class `rational` and output another member of our class. For the present, we will not worry about reducing our result to its simplest form; we will add this capability in a minute.

- Should we use a function or a subroutine?

- Recall that to add two rational numbers we simply find a common denominator and combine, i.e.,

\[
\frac{p}{q} + \frac{r}{s} = \frac{ps + rq}{qs}
\]

- To define a new member of our class we need to specify its numerator and denominator. If this member is found as the sum of two other members of our class, we need to define its numerator and its denominator from the formula above. Consequently, our function could look like:
function add_two_rationals (a, b )  result (c)
  type(rational), intent(in) ::  a, b
  type(rational) ::  c
  c % num = a % num * b % den + b % num * a% den
  c % den = a % den * b % den
end function add_two_rationals

• Then to invoke this routine we could simply type
  rat = add_two_rationals ( a, b)
  where a, b, rat have been defined as members of our class rational.

• However, wouldn’t it be easier if we could write
  rat = a + b

• Operator overloading allows us to do this.
• Note that we have already encountered operator overloading and we haven’t realized it.

• For example, the usual operator + responds differently when we write \( c = a + b \) if \( a, b, c \) are integers, or reals or complex numbers.

• Operator overloading allows us to redefine this operator + based upon a data structure we have defined (such as our rational
To accomplish this each operator that you want to use has to be defined, or overloaded, for each data type. This is accomplished through the use of the **interface operator** construct and is somewhat similar to our example of **function overloading** when we used the generic name `compute_area` to call `compute_area_circle` or `compute_area_rectangle` depending on the specific data type in the argument. Also we used function overloading when we called `print_date` or `print_employee_info` with the generic call `print_info`.

In our case we can use the following syntax to overload `+` to use our function to add two rational numbers. It is only invoked when the objects we are adding are members of our defined data structure `rational`. As before, we will put it in our module that defines our data structure using this particular overloading:

```plaintext
interface operator (+)
    module procedure add_two_rationals
end interface
```
• Of course operator overloading is not just restricted to +.

• We can overload other operators such as -, *, /, >, <, ==.

• In addition, we can overload an assignment operator =. In this case we use the word assignment instead of operator.

```plaintext
interface assignment (= )
    module procedure set_equal_integer
end interface
```

• You can have more than one operator or assignment procedure but each one must take different argument types. For example, we can overload the operator + to use the routine add_two_rationals and also overload the same operator to add_two_vectors (which we will do soon). Then in the main program which uses both of these modules we can write c = a + b; then depending if a, b, c are vectors or rationals, the main program will call the appropriate routine.
module class_rationals

use common_data

implicit none

private ::  list routines that are not accessible to other users

module type rational

    integer :: num

    integer :: den

end type rational

interface operator (+)

    module procedure add_two_rationals

end interface operator (+)

end interface

contains

our subprograms

end module class_rationals

In the main program if we declare

type(rational) :: a, b, c

then when we have the statement

c = a + b

it will automatically call the function \texttt{add\_two\_rationals} with the first input as \texttt{a}, the second as \texttt{b} and the output in \texttt{c}.
Classwork

Define a data structure called `manager` which has attributes:

- all employee information (don’t enter separately just declare that data structure)
- annual salary (real number)
- whether manager is assistant manager or manager (character)
- date promoted to manager status

Write a routine to print out a manager’s information and add statements to call it with the generic name `print_info`. Add statements to the code `test_employee` to make sure everything is working.