Fortran classes and objects

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: August 28, 2022

1. Classes and objects

Fortran classes are based on `type` objects.
Similarities and differences with C++.

- Same `%` syntax for specifying data members and methods.
- Data and functions declared separately.
- Object itself as extra parameter.

All will become clear . . .

2. Object is type with methods

You define a type as before, with its data members, but now the type has a `contains` for the methods:

```
Module multmod

  type Scalar
     real(4) :: value
   contains
     procedure,public :: &
          printme,scaled
  end type Scalar

contains ! methods
```

3. Object methods

Method call similar to C++

Code:

```
Program Multiply
  use multmod
  implicit none

  type(Scalar) :: x
  real(4) :: y
  x = Scalar(-3.14)
  call x%printme()
  y = x%scaled(2.)
  print '(f7.3)',y

end Program Multiply
```

Output

```
[objectf] mult1:

The value is -3.140
 -6.280
```

4. Method definition

Note the extra first parameter:
which is a `Type` but declared here as *`Class`*:

```
subroutine printme(me)
  implicit none
  class(Scalar) :: me
  print '("The value is",f7.3)',me%value
end subroutine printme
function scaled(me,factor)
  implicit none
  class(Scalar) :: me
  real(4) :: scaled,factor
  scaled = me%value * factor
end function scaled
```

5. Class organization

- You're pretty much forced to use `Module`
- A class is a `Type` with a `contains` clause
  followed by `procedure` declaration
- Actual methods go in the `contains` part of the module
- $\Rightarrow$ First argument of method is the object itself. $\Leftarrow$

6. Similarities and differences

	C++	Fortran
Members	in the object	in the 'type'
Methods	in the object	interface: in the type implementation: in the module
Constructor	default or explicit	none
object itself	'this'	first argument
Class members	global variable	accessed through first arg
Object's methods	period	percent

7. Point program

```
Module PointClass
  Type,public :: Point
     real(8) :: x,y
   contains
     procedure, public :: &
        distance
  End type Point
contains
  !! ... distance function ...
  !! ...
End Module PointClass
```

```
Program PointTest
  use PointClass
  implicit none
  type(Point) :: p1,p2

  p1 = point(1.d0,1.d0)
  p2 = point(4.d0,5.d0)

  print *,"Distance:",&
      p1%distance(p2)

End Program PointTest
```

Exercise 1

Take the point example program and add a distance function:

```
Type(Point) :: p1,p2
! ... initialize p1,p2
dist = p1%distance(p2)
! ... print distance
```

You can base this off the file `pointexample.F90` in the repository

Exercise 2

Write a method add for the `Point` type:

```
Type(Point) :: p1,p2,sum
! ... initialize p1,p2
sum = p1%add(p2)
```

What is the return type of the function add?

Operator overloading

8. Define operators on classes

```
Type(X) :: x,y,z

! function syntax:
x = y%add(z)

! operator syntax
x = y+z
```

Code looks closer to math formulas

9. Example class

For purposes of exposition, let's make a very simple class:

```
Type,public :: ScalarField
   real(8) :: value
 contains
   procedure,public :: set,print
   procedure,public :: add
End type ScalarField
```

10. Methods just for testing

```
subroutine set(v,x)
  implicit none
  class(ScalarField) :: v
  real(8),intent(in) :: x

  v%value = x
end subroutine set

subroutine print(v)
  implicit none
  class(ScalarField) :: v

  print '(f7.4)', v%value
end subroutine print

call u%set(2.d0)
call v%set(1.d0)
!  z = u%add(v)
z = u+v
```

11. Addition function

```
function add(in1,in2) result(out)
  implicit none
  class(ScalarField),intent(in) :: in1
  type(ScalarField),intent(in) :: in2
  type(ScalarField) :: out

  out%value = in1%value + in2%value
end function add
```

Parameters need to be `Intent(In)`

12. Operator definition

Interface block:

```
interface operator(+)
   module procedure add
end interface operator(+)
```

Exercise 3

Extend the above example program so that the type stores an array instead of a scalar.

Code:

```
integer,parameter :: size = 12

Type(VectorField) :: u,v,z

call u%alloc(size)
call v%alloc(size)
call u%setlinear()
call v%setconstant(1.d0)
!  z = u%add(v)
z = u+v
call z%print()
```

Output

```
[geomf] field:

 2.0000  3.0000
    4.0000  5.0000
    6.0000  7.0000
    8.0000  9.0000
    10.0000 11.0000
    12.0000 13.0000
```

You can base this off the file `scalar.F90` in the repository