

# Derived Types and Modules in Fortran

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2020

# Types

# Structures: type

- Fortran has structures too.
- Structures are a derived type: you can create variables of that type, but it's not a built-in type.
- Fortran derived types are declared in a type definition;
- the type keyword is also used for making structure variables.

# Type declaration

Type name / End Type name block.

Member declarations inside the block:

```
type mytype
  integer :: number
  character :: name
  real(4) :: value
end type mytype
```

# Creating a type structure

Declare a type variable in the main program:

```
Type(mytype) :: object1,object2
```

Initialize with type name:

```
object1 = mytype( 1, 'my_name', 3.7 )
```

Copying:

```
object2 = object1
```

# Member access

Access structure members with %

```
Type(mytype) :: typed_object  
typed_object%member = ....
```

# Example

```
type point  
  real :: x,y  
end type point
```

```
type(point) :: p1,p2  
p1 = point(2.5, 3.7)  
  
p2 = p1  
print *,p2%x,p2%y
```

Type definitions can go in the main program  
(or use a module; see later)

# Structures as procedure argument

Structures can be passed as procedure argument, just like any other datatype. This example:

- Takes a structure of `type(point)` as argument; and
- returns a `real(4)` result.
- The structure is declared as `intent(in)`.

Function with structure  
argument:

```
real(4) function length(p)
  implicit none
  type(point), intent(in) :: p
  length = sqrt( &
    p%x**2 + p%y**2 )
end function length
```

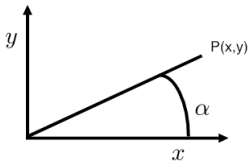
Function call

```
print *, "Length:", length(p2)
```



# Exercise 1

Add a function *angle* that takes a *Point* argument and returns the angle of the *x*-axis and the line from the origin to that point.



Your program should read in the *x*, *y* values of the point and print out the angle in radians.

Bonus: can you print the angle as a fraction of  $\pi$ ? So

$$(1, 1) \Rightarrow 0.25$$

## Exercise 2

Write a program that has the following:

- A type `Point` that contains real numbers `x,y`;
- a type `Rectangle` that contains two `Points`, corresponding to the lower left and upper right point;
- a function `area` that has one argument: a `Rectangle`.

Your program should

- Accept two real numbers on one line, for the bottom left point;
- similarly, again on one line, the coordinates of the top right point; then
- print out the area of the (axi-parallel) rectangle defined by these two points.

# Modules

# Module definition

Modules look like a program, but without main (only 'stuff to be used elsewhere'):

```
Module definitions
  type point
    real :: x,y
  end type point
  real(8),parameter :: pi = 3.14159265359
contains
  real(4) function length(p)
    implicit none
    type(point),intent(in) :: p
    length = sqrt( p%x**2 + p%y**2 )
  end function length
end Module definitions
```

Note also the numeric constant.

# Module use

Module imported through use statement;  
comes before implicit none

**Code:**

Program size

```
use definitions
implicit none
```

```
type(point) :: p1,p2
p1 = point(2.5, 3.7)
```

```
p2 = p1
print *,p2%x,p2%y
print *,"length:",length(p2)
print *,2*pi
```

end Program size

**Output**

[structf] typemod:

```
2.50000000      3.70000005
length:  4.46542263
6.2831854820251465
```

# Module use

```
Program ModProgram
  use FunctionsAndValues
  implicit none

  print *, "Pi is:", pi
  call SayHi()

End Program ModProgram
```

Also possible:

```
Use mymodule, Only: func1, func2
Use mymodule, func1 => new_name1
```

## Exercise 3

Take exercise 2 and put all type definitions and all functions in a module.

# Turn it in!

- If you have compiled your program, do:  
`coe_areaf yourprogram.F90`  
where 'yourprogram.F90' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:  
`coe_areaf -s yourprogram.F90`  
where the `-s` flag stands for 'submit'.
- If you don't manage to get your code working correctly, you can submit as incomplete with  
`coe_areaf -i yourprogram.F90`
- Use the `-d` debug flag for more information.

For bonus points, use a module.



# Separate compilation of modules

Suppose program is split over two files:

`theprogram.F90` and `themodule.F90`.

- Compile the module: `ifort -c themodule.F90`; this gives
- an object file (extension: `.o`) that will be linked later, and
- a module file `modulename.mod`.
- Compile the main program:  
`ifort -c theprogram.F90` will read the `.mod` file; and finally
- Link the object files into an executable:  
`ifort -o myprogram theprogram.o themodule.o`  
The compiler is used as linker: there is no compiling in this step.

Important: the module needs to be compiled before any (sub)program that uses it.