# Derived Types and Modules in Fortran

Kevin Schmidt, Susan Lindsey, Charlie Dey

Spring 2019

# Structures

# Structures: `type`

The Fortran name for structures is `type` or *derived type*.

# Type definition

`Type name` / `End Type` block.
Variable declarations inside the block

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

# Creating a type structure

Declare a type object 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
type_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

# Types as subprogram argument

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

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

# Exercise 1

Define a type `Point` that contains real numbers `x,y`.
Define a type `Rectangle` that contains two `Points`.
Write a function `area` that has one argument, a `Rectangle`.

# Modules

# Module definition

Modules look like a program, but without executable code:

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

# Module use

Module imported through `use` statement;
comes before `implicit none`

```
Program size
  use definitions
  implicit none

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

  p2 = p1
  print *,p2%x,p2%y

end Program size
```

# Exercise 2

Take exercise 1 and put all type definitions and all functions in a module.

# Separate compilation of modules

Suppose program is split over `theprogram.F90` and `themodule.F90`.

- `icpc -c themodule.F90`; this gives
- an *object file* that will be linked later, and
- a `.mod` file (with the name of the module, not of the file);
- `icpc -c theprogram.F90` will read the `.mod` file; and finally
- `icpc -o myprogram theprogram.o themodule.o` uses the compiler as *linker* to form the executable.

The module needs to be compiled before the program.