Derived Types and Modules in Fortran

Victor Eijkhout, Susan Lindsey

Fall 2022
`last formatted: November 2, 2022`

Types

1. Structures: `type`

- Fortran has structures similar to C:
  bundle variables – of different types.
- Structures are a derived type: you can create variables of that type, but it's not a built-in type.
- Fortran keyword for derived types is (confusingly) `Type`

2. 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
```

Type definitions go before executable statements.

3. Creating / initializing type variables

Declare type variables in the main program:

```
Type(mytype) :: struct1,struct2
```

Initialize with type name:

```
struct1 = mytype( 1, 'my_name', 3.7 )
```

Copying:

```
struct2 = struct1
```

4. Member access

Access structure members with %
(compare C++ dot-notation)

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

5. 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
```

6. Structures as procedure argument

Structures can be passed as procedure argument, just like any other datatype. In this example the function *length*:

- 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:

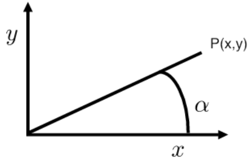
```
real(4) function length(p)
  implicit none
  type(point),intent(in) :: p
  length = sqrt( &
       p%x**2 + p%y**2 )
end function length
```

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 π? So

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

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

Exercise 2

Write a program that has the following:

- A type *Point* that contains real numbers *x*, *y*;
- a type *Rectangle* that contains two *Point*s, 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.

7. Definitions

Type definition:

```
type var
   character(len=20) :: id
   integer :: value
end type var
```

Exercise 3

The following main program should give the corresponding output:

Code:

```
print *,x
print *,y
z = varadd(x,y)
print *,z
a = varmult(z,2)
print *,a
```

Output:

```
 x                                     1
 y                                     2
 (x)+(y)                               3
 2((x)+(y))                            6
```

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

Turn it in!

- If you have compiled your program, do:

  `coe_varf 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_varf -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_varf -i yourprogram.F90`
- Use the `-d` debug flag for more information.

For bonus points, use a module.

Modules

8. What are modules

Programming is about introducing abstractions.

- Functions are good.
- … but in a `Contains` section they can only be used in one program.
- ⇒ Put useful functions in a module and use that module anywhere.
- 'Software library'

9. Module definition

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

```
Module geometry
  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 geometry
```

Note also the numeric constant.

10. Example use

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

Code:

```
Program size
  use geometry
  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:

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

Exercise 4

Move the *var* type definition and the routines *varadd*, *varmult* into a module. Add the routines *islower*, *isdigit* from the character exercises.

Exercise 5

Write a loop that accepts character input, and only prints out what kind of character was encountered.

Use a module for the functions that recognize the characters.

11. Stack definition

```
type var
   character :: id = "."
   character(len=20) :: expression
   integer :: value
end type var

type(var),dimension(10) :: stack
integer :: stackpointer=0
```

Exercise 6

Make your event loop accept digits, creating a new entry, and lowercase characters, naming the top entry:

Code:

```
   else if ( islower(input) ) then
!! ...
          stack(stackpointer)%id = input
```

Output:

```
Inputs: 1 x 2 y 0
id:. expr=1 val=1;
id:x expr=1 val=1;
id:x expr=1 val=1; id:. expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
```

Exercise 7

Add a clause to your event loop:

Code:

```
else if ( isop(input) ) then
```

Output:

```
Inputs: 1 2 + z 0
id:. expr=1 val=1;
id:. expr=1 val=1; id:. expr=2 val=2;
id:. expr=(1)+(2) val=3;
id:z expr=(1)+(2) val=3;
```

Advanced module topics

12. Module use syntax

`Use` statement placed before `Implicit`

```
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
```

13. 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.