Numerical Modelling in FORTRAN day 9

Paul Tackley, 2020

Today's Goals

Focus on some features introduced since Fortran 95, particularly related to derived types (=>"objects"):

1. Parameterized derived types
2. Type extension & polymorphism
3. Type-bound procedures
4. Data hiding

Exercises to practice & learn these features.

Debugging with Intel fortran

- **ifort** allows some non-standard language usage that can cause compilation errors on other compilers (like gfortran).
- To be warned about these, use e.g.
  - ifort –stand f95 program.f90
  - ifort –stand f18 program.f90
  - Possible standards: f90, f95, f03, f08, f18
  - Probably best to use the latest! (f18)

Parameterized derived types

- Types in which the kind and length can be specified at run time (when they are dummy arguments).
- Use the keywords ‘kind’ and ‘len’ to indicate these.

```
program parameterised_type
  use iso_fortran_env  ! contains various useful things
  implicit none

  type:: matrix(k,n,m)
     integer, kind :: k = kind(0.)  ! Set default values
     integer, len  :: n=3, m=3
     real(kind=k)  :: value(n,m)
  end type matrix

  type(matrix(real64,10,20)):: mat  ! real64 is in iso_fortran_env
  type(matrix(n=5,m=5)):: mat2      ! use default kind

  call random_number(mat2%value)
  call matprint(mat2)

contains

  subroutine matprint(mat_in)
    type(matrix(n=*,m=*)) :: mat_in ! assumed size (*)
    print*,mat_in%k
    print*,mat_in%n
    print*,mat_in%m
    print*,mat_in%value
  end subroutine matprint

end program parameterised_type
```

```
           4
           5
           5
 0.159648478       0.837237418       0.775771737       0.103784740       0.835458457
  4.42287922E-02   0.755849242       0.609476864       0.151317716       0.964308679
 0.827501655       0.117429316       0.662792087       0.884603441       0.543005526
 0.449754834       0.181883693       0.141135156       0.672917724       0.683630705
 0.779614866       0.157992601       0.562648058       0.491638660       0.251410484
```

Analysis

- Setting **default values** is a good idea; then passing actual values is optional
- Use of **assumed size** (*) in procedures: the size will automatically be known
- Use of iso_fortran_env: contains useful things such as the kind numbers for real32, real64, int32, int64, etc.

Type extension

- Creates a new derived type by extending an existing one.
- The new type has all components of the existing type (the **parent**) plus specified additional ones.

Similar to in day 5

Extends person with 2 additional things

Note new syntax

Parent is inherited

```
program TypeExtension1
  implicit none

  type person
     character(len=20):: Name
     integer :: BirthYear
     integer :: nChildren
  end type person

  type, extends(person) :: student
     character(len=10):: Degree
     integer :: StartYear
  end type student

  type(person):: aperson
  type(student):: astudent

  aperson = person('John',1940,2)
  astudent= student('Jill',1998,0,'MSc',2019)
  print*,astudent%Degree                    MSc
  print*,astudent%Name                      Jill
  print*,astudent%person%Name               Jill

end program TypeExtension1
```

Analysis

- Use of type,extends(...)
- Components in the parent can be accessed directly or via the parent, e.g.
  - astudent%Name or
  - astudent%person%Name
- Extended type can be further extended

Extended types can be further extended

Different extension of person

```
module persontypes
  implicit none
  type person
     character(len=20) :: Name
     integer :: BirthYear
     integer :: nChildren
  end type person

  type, extends(person) :: student
     character(len=10):: Degree
     integer :: StartYear
  end type student

  type, extends(student) :: PhD_student
     character(len=50):: ResearchProject
  end type PhD_student

  type, extends(person) :: professor
     integer :: PhDYear
     character(len=50):: Department
  end type professor

end module persontypes
```

Polymorphic variables

- A variable whose type can vary at run time
- Must be a pointer, allocatable variable or dummy argument
- Declare using class instead of type
- Can be of the declared type or any of its extensions
- Declared type = the type it was declared to be
- Dynamic type = the type it presently is

```
program Polymorphic1
  use persontypes
  implicit none
  type(person) :: p1=person('Adam',1980,1)
  type(PhD_student) :: p2=PhD_student('Eve',1995,1,'PhD',2017,'Mars')
  type(professor) :: p3=professor('Margaret',1960,3,1990,'Earth Sciences')

  call Print_Name(p1)
  call Print_Name(p2)
  call Print_Name(p3)

contains

  subroutine Print_Name(p_in)
    class(person),intent(in):: p_in
    print*,'Name = ',p_in%Name
  end subroutine Print_Name

end program Polymorphic1
```

```
Name = Adam
Name = Eve
Name = Margaret
```

Polymorphic dummy argument variable

Polymorphic variable: limitation

- The polymorphic variable can only access components of the parent (the *declared* type), not the extra components in the extended types
- Solution: Use a select type construct

```
program SelectType
  use persontypes
  implicit none
  type(person) :: p1=person('Adam',1980,1)
  type(PhD_student) :: p2=PhD_student('Eve',1995,1,'PhD',2017,'Mars')
  type(professor) :: p3=professor('Margaret',1960,3,1990,'Earth Sciences')

  call Print_Info(p1)
  call Print_Info(p2)
  call Print_Info(p3)

contains

  subroutine Print_Info(p_in)
    class(person),intent(in):: p_in
    print'(a,$)',trim(p_in%Name)
    select type (p_in)
    type is (professor)
       print*,' is a professor in ',p_in%Department
    class is (student)
       print*,' is a student working towards a ',p_in%Degree
    class default
       print*,' was born in ',p_in%Birthyear
    end select
  end subroutine Print_Info

end program SelectType
```

```
Adam  was born in          1980
Eve  is a student working towards a PhD
Margaret  is a professor in Earth Sciences
```

Useful intrinsic functions

- same_type_as(a,b): Logical .true. if a & b have the same dynamic type.
- extends_type_of(a,b): Logical .true. If the dynamic type of a is an extension of the dynamic type of b.

Type-bound procedures

- Procedures are “bound” with a derived data type.
- Idea is to create an “object” containing all data and the procedures to set/perform operations on/access/finalise that data.
- Can only be used with variables of the data type they are defined in.
- Invoked using %, as with variables in the type.
- Contained within the type definition.

Example:
point routines
from last time

'pass'
means type
is passed
as 1st arg.

Use "class"

Access using %

```
program test
  use coords
  type(point):: p1,p2,p3
  p1%x=1.2; p1%y=0. ; p1%z=3.1
  p2%x=0. ; p2%y=1.2; p2%z=1.7

  p3 = p1%pointplus (p2)        ; print*,p3
  p3 = p1%pointminus(p2)        ; print*,p3
  d  = p1%absvec()              ; print*,d
  d  = p1%pointseparation(p2)   ; print*,d

end program test
```

```
module coords
  implicit none

  type point                ! derived type
     real:: x,y,z
   contains
     procedure,pass :: pointplus, pointminus
     procedure,pass :: pointseparation, absvec
  end type point

contains

  type(point) function pointplus(this,b)   ! for +
    class(point),intent(in):: this,b
    pointplus%x = this%x + b%x
    pointplus%y = this%y + b%y
    pointplus%z = this%z + b%z
  end function pointplus

  type(point) function pointminus(this,b)  !  for -
    class(point),intent(in):: this,b
    pointminus%x = this%x - b%x
    pointminus%y = this%y - b%y
    pointminus%z = this%z - b%z
  end function pointminus

  real function pointseparation(this,b) ! for .distance.
    class(point),intent(in):: this,b
    pointseparation = sqrt(  &
          (this%x-b%x)**2+(this%y-b%y)**2+(this%z-b%z)**2)
  end function pointseparation

  real function absvec(this)  ! distance from origin
    class(point),intent(in):: this
    absvec = sqrt(this%x**2+this%y**2+this%z**2)
  end function absvec

end module coords
```

Analysis

- “pass” means that the typed variable is passed as the 1st argument. (default)
- “nopass” is also possible.
- Access procedures using %, just like variables in the type.
- Procedure arguments must be declared class(...) not type(...)

Improvement: rename procedures

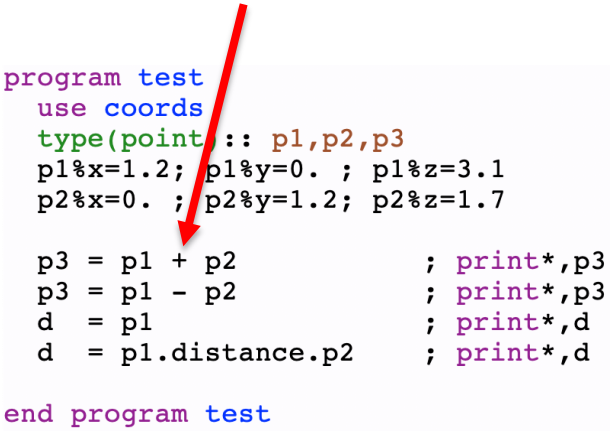
Use more compact names in the ‘type’ using =>

```
type point
   real:: x,y,z
 contains
   procedure,pass :: plus      => pointplus
   procedure,pass :: minus     => pointminus
   procedure,pass :: distance  => pointseparation
   procedure,pass :: magnitude => absvec
end type point
```

```
p3 = p1%plus (p2)
p3 = p1%minus(p2)
d  = p1%magnitude()
d  = p1%distance(p2)
```

Improvement: use generic/overloaded operators

Use +/- etc. like last week



```
program test
  use coords
  type(point) :: p1,p2,p3
  p1%x=1.2; p1%y=0. ; p1%z=3.1
  p2%x=0. ; p2%y=1.2; p2%z=1.7

  p3 = p1 + p2            ; print*,p3
  p3 = p1 - p2            ; print*,p3
  d  = p1                 ; print*,d
  d  = p1.distance.p2     ; print*,d

end program test
```

```
module coords
  implicit none

  type point
     real:: x,y,z
   contains
     procedure :: pointplus, pointminus, pointseparation
     procedure,pass(this) :: absvec
     generic :: operator(+) => pointplus
     generic :: operator(-) => pointminus
     generic :: operator(.distance.) => pointseparation
     generic :: assignment(=) => absvec
  end type point

contains

  type(point) function pointplus(this,b)   ! for +
    class(point),intent(in): this,b
    pointplus%x = this%x + b%x
    pointplus%y = this%y + b%y
    pointplus%z = this%z + b%z
  end function pointplus

  type(point) function pointminus(this,b)  !  for -
    class(point),intent(in):: this,b
    pointminus%x = this%x - b%x
    pointminus%y = this%y - b%y
    pointminus%z = this%z - b%z
  end function pointminus

  real function pointseparation(this,b) ! for .distance.
    class(point),intent(in):: this,b
    pointseparation = sqrt(  &
         (this%x-b%x)**2+(this%y-b%y)**2+(this%z-b%z)**2)
  end function pointseparation

  subroutine absvec(a,this)  ! for = (distance
    real,intent(out):: a  !  from origin)
    class(point),intent(in):: this
    a = sqrt(this%x**2+this%y**2+this%z**2)
  end subroutine absvec

end module coords
```

Analysis

- First need to list the routines using procedure, then use generic :: to overload +/-/= or create a new operator
- Complexity with = (absvec) because the input must be the 2nd argument but “pass” normally passes type as the 1st argument
- Additional procedures could be added to handle different data types and further overload +/-/= etc.

Dealing with assignment (=) overload

```
type point
   real:: x,y,z
 contains
   procedure :: pointplus, pointminus, pointseparation
   procedure,pass(this) :: absvec
   generic :: operator(+) => pointplus
   generic :: operator(-) => pointminus
   generic :: operator(.distance.) => pointseparation
   generic :: assignment(=) => absvec
end type point
```

Specifies argument to pass

2nd argument

```
subroutine absvec(a,this)  ! for = (distance
  real,intent(out):: a  !  from origin)
  class(point),intent(in):: this
  a = sqrt(this%x**2+this%y**2+this%z**2)
end subroutine absvec
```

Type extension with type-bound procedures

- Extended types can add new bound procedures or overwrite existing ones.
- Polymorphic variables can only access the procedures in the parent (superclass) type.
- Therefore, best to include all relevant procedures in the parent type even if they don’t do anything.

Data hiding

- By default, everything in a module is visible to a useing routine.
- But many things (e.g. *pointplus*, *absvec*) should not be directly accessed from outside the module. They should be hidden.
- Use attributes public, private & protected to accomplish this.

Public, private, protected

- Public: fully accessible to a useing routine.
- Private: hidden from a useing routine.
- Protected: read-only by a useing routine (cannot be modified).
- These can be specified for
  - Each variable or procedure
  - Lists of variables & procedures
  - All variables & procedures

Public, private, protected

```
module coords
  implicit none

  private
  public :: point

  type point
     real :: x,y,z
   contains
     procedure,private :: pointplus, pointminus, pointseparation
     procedure,private,pass(this) :: absvec
     generic,public :: operator(+) => pointplus
     generic,public :: operator(-) => pointminus
     generic,public :: operator(.distance.) => pointseparation
     generic,public :: assignment(=) => absvec
  end type point

contains
```

Typical usage

- Start module with “private” to make everything private by default.
- Then list things that should be public.
- To be completely clear, include private/public/protected in all declarations.

Exercise 1

Improve **type point** in the provided **coords** module in the following ways and write a suitable main program to test everything:

(i) Enhance assignment(=) so that it can return either 32- or 64-bit real numbers. That is, the statement

```
d=p1
```

should work with **d** being either 32-bit or 64-bit.

In order to do this, 2 versions of subroutine absvec() are needed: one of (kind=real32) and one of (kind=real64). List them both in the **contains** area of type point.

(Note that d=p1.distance.p2 will already work with 32-bit or 64-bit **d** because the type of the function arguments does not change.)

Exercise 1 continued

(ii) Add two more operators:

.dot. Calculates the dot product of two vectors held in type point
(result: real. Both precisions will work without effort)

.cross. Calculates the cross product of two vectors held in type point
(result: type point)

(iii) Add ”*” and “/” operators to allow points to be multiplied or divided by real32 or real64 scalars. Make sure that ”*” works in either order, i.e.

p2=p1*b and p2=b*p1 should both work (b is real32 or 64, p1&p2 are point)

For divide, only p1/b needs to be implemented.

(iv) Declare a new **type** that **extends** type(point) by holding the temperature value at each point. Add new type bound procedures Tplus and Tminus to add and subtract the temperatures of two variables of this type, e.g. dT=p1%Tplus(p2) .

Exercise 1 continued

Notes:

Add (i)-(iv) cumulatively to the same module and program and hand in only one working program that incorporates all improvements.

An extended type as in (iv) is normally defined in a separate module. This is definitely necessary when overriding procedure(s), although this is not done here.

Make sure all of these work:

```
real*4 r4
real*8 r8
type(point):: p1,p2,p3
type(pointT):: pT1,pT2
```

```
r4 = p1
r8 = p1
r4 = p1.dot.p2
p3 = p1.cross.p2
p3 = p1 * r4
p3 = p1 * r8
p3 = r4 * p1
p3 = r8 * p1
p3 = p1 / r4
p3 = p1 / r8
r4 = pT1%Tplus(pT2)
r4 = pT1%Tminus(pT2)
```

Exercise 2

(from Chapman, Fortran for Scientists and Engineers)

12-2. Create a derived data type called "polar" to hold a complex number expressed in polar (z, θ) format as shown in Figure 12-8. The derived data type will contain two components, a magnitude z and an angle θ, with the angle expressed in degrees. Write two functions that convert an ordinary complex number into a polar number, and that convert a polar number into an ordinary complex number.

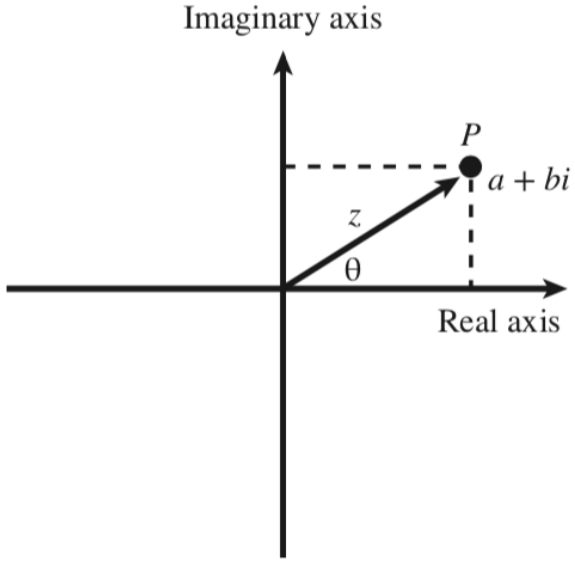


FIGURE 12-8
Representing a complex number in polar coordinates.

12-3. If two complex numbers are expressed in polar form, the two numbers may be multiplied by multiplying their magnitudes and adding their angles. That is, if $P_1 = z_1 \angle \theta_1$ and $P_2 = z_2 \angle \theta_2$, then $P_1 \cdot P_2 = z_1 z_2 \angle \theta_1 + \theta_2$. Write a function that multiplies two variables of type "polar" together using this expression and returns a result in polar form. Note that the resulting angle θ should be in the range $-180° \leq \theta \leq 180°$.

12-4. If two complex numbers are expressed in polar form, the two numbers may be divided by dividing their magnitudes and subtracting their angles. That is, if $P_1 = z_1 \angle \theta_1$ and $P_2 = z_2 \angle \theta_2$, then $\dfrac{P_1}{P_2} = \dfrac{z_1}{z_2} \angle \theta_1 - \theta_2$. Write a function that divides two variables of type "polar" together using this expression and returns a result in polar form. Note that the resulting angle θ should be in the range $-180° \leq \theta \leq 180°$.

12-5. Create a version of the `polar` data type with the functions defined in Exercises 12-2 through 12-4 as bound procedures. Write a test driver program to illustrate the operation of the data type.

Hand in by 7 December

- Source files that compile and run (module + main program to test it)
- Make sure it compiles with
  - “gfortran –fcheck=all –finit-real=snan –ffpe-trap=invalid,zero,overflow –O0”