Lecture IV: Derived data types. Yet more about procedures.

Outline of the first part

- Recalling Fortran built-in data types
- Rationale behind derived data types
- Data type declaration and visibility

Fortran built-in types

- Standard Fortran already supports a wide variety of fundamental data types to represent integers, floating point numbers (real), truth values (logical) and variable length character strings
  - Each of these built-in types may have declared as multi-dimensional array
- Representation of reals and integers can be controlled through kind parameter (e.g. 8 or 4)
  - Numerical precision
  - Memory consumption, CPU register considerations

A few words about numerical precision

- The variable representation method (precision) may be declared using the KIND statement

```
! selected_int_kind(r)
! selected_real_kind(p)
! selected_real_kind(p,r)

integer, parameter :: short=selected_int_kind(4)
integer, parameter :: double=selected_real_kind(12,100)
integer (kind=short) :: index
real (kind=double) :: x,y,z
complex (kind=double) :: c

x=1.0_double; y=2.0_double * acos(x)
```

Integer between $-10^r < n < 10^r$

Real number accurate to *p* decimals

A real number between $-10^{100} < x < 10^{100}$, accurate to 12 decimals

Numerical precision

```
PROGRAM Precision_Test
implicit none

integer, parameter :: sp = selected_real_kind(6,30), &
                      dp = selected_real_kind(10,200)
real(kind=sp) :: a
real(kind=dp) :: b
write(*,*) sp, dp, kind(1.0), kind(1.0_dp)
write(*,*) kind(a), huge(a), tiny(a), range(a), precision(a)
write(*,*) kind(b), huge(b), tiny(b), range(b), precision(b)

end program precision_test
```

output:

```
 4  8  4  8
 4 3.4028235e+38 1.1754944e-38 37 6
 8 1.797693134862316e+308 2.225073858507201e-308 307 15
```

Module ISO_FORTRAN_ENV

```
MODULE prec
    USE ISO_FORTRAN_ENV, ONLY : INT32, INT64, REAL32, REAL64
    IMPLICIT NONE
    PRIVATE
    INTERGER, PARAMETER :: i4 = INT32 &
                           i8 = INT64 &
                           r4 = REAL32 &
                           r8 = REAL64
    PUBLIC :: i4, i8, r4, r8
END MODULE prec
```

Numerical precision

Other intrinsic functions related to numerical precision

KIND(A)	Returns the kind of the supplied argument
TINY(A)	The smallest positive number
HUGE(A)	The largest positive number
EPSILON(A)	The smallest positive number added to 1.0 returns a number just greater than 1.0
PRECISION(A)	Decimal precision
DIGITS(A)	Number of significant digits
RANGE(A)	Decimal exponent
MAXEXPONENT(A)	Largest exponent (of the kind(A))
MINEXPONENT(A)	Smallest exponent (of the kind(A))

What is derived data type?

- Derived data type is a data structure composed of built-in data types and possibly other derived data types
  - Equivalent to structs in C programming language
- Derived type is defined in the variable declaration section of programming unit
  - Not visible to other programming units
    - Unless defined in a module and used via USE clause, which is most often the preferred way

Derived data types – rationale

- Properly constructed data types make the program more readable, lead to clean interfaces and less errors
- Variables used in the same context should be grouped together, using modules and derived data types
- Data layout computational efficiency should be kept in mind when diving into object oriented programming in Fortran (or any other language)

Data type declaration

- Type declaration

```
TYPE playertype
  CHARACTER (LEN=30) :: name
  INTEGER :: number, goals, assists
END TYPE playertype
```

- Declaring variables using a derived data type

```
TYPE(playertype) :: ville, pekka
TYPE(playertype), DIMENSION(30) :: players
```

Accessing data types

- Initialization

```
ville%name = ‘Ville Nieminen'
ville%number = 17
ville%goals = 10
ville%assists = 8
```

- Alternatively

```
ville = playertype(‘Ville Nieminen', 17, 10, 8)
```

- Vector of derived data type: element-wise addressing

```
players(1)%name = 'Pekka Saravo'
players(1)%number = 6
players(1)%goals = 2
players(1)%assists = 4
```

Nested derived types

- Declaration of a derived type using another derived type

```
TYPE hockeyteam
  CHARACTER (LEN=80) :: name
  TYPE(playertype) :: players(30)
  TYPE(goalietype) :: goalies(3)
END TYPE hockeyteam
```

- Declaring variables:

```
TYPE(hockeyteam) :: tappara, ilves, karpat
```

- Initialization / access example:

```
tappara%name = 'Tappara'
 tappara%players(2)%name = 'Ville Nieminen'
 tappara%players(2)%number = 17
```

Visibility of derived data types

- When declared in the same programming unit derived data types are visible to that unit only
  - and subunits under CONTAINS statement
- When declared in a module unit, a derived data type can be accessed outside the module through the USE statement

Outline of the second part

- Pointers to arrays
- Generic procedures
- Operator overloading

Pointers to arrays

- The POINTER attribute enables to create array (or scalar) aliasing variables
  - It can refer to a variable that is has either an attribute TARGET
  - It can also refer to another variable with the pointer attribute
- A pointer variable can also be a sole variable itself
  - A desired shape given with the ALLOCATE statement
- C programmers: ”pointer” has a different meaning in C and Fortran

Pointers to arrays

- A POINTER can refer to an already allocated memory region

```
integer, pointer :: p_x(:) => null()
integer, target :: x(1000)
...
p_x => x
p_x => x ( 2 : 300 )
p_x => x ( 1 : 1000 : 5 )
...
p_x(1) = 0
nullify(p_x)
```

Pointer, initialized to point to nothing

Now p_x equivalent to x

Pointers provide a neat way for array sections

This would change also x(1) to 0

Disconnects p_x's connection to x

Pointers to arrays

- Examples with a 2D array

```
integer :: n=100, m=200
real, pointer :: p_mat (:,:), p_vec(:), p_diag(:)
real, target :: mat (n,m)

p_mat => mat
p_mat => mat (1:50 , 1:50 )
p_mat => mat (10:100:10 , 10::5)
p_vec(1:n*m) => mat
p_diag => p_vec(::n+1)
```

Pointers to arrays

- Whether a POINTER points to anything, use ASSOCIATED – function to check :

```
real, pointer :: p_mat (:,:) => null ()
...
if ( associated (p_mat) ) then
    print *,’points to something’
else
    print *,’points to nothing’
end if
```

Generic procedures

- In Fortran, a procedure must know the data types of its arguments as well as local variables
- Procedures which perform similar actions but for different data types can be defined as *generic procedures*
  - Procedures are called using the *generic name* and compiler uses the correct procedure based on the argument number, type and dimensions
    - Compiler error if no matching procedure found
- Generic name is defined in an INTERFACE section

Generic procedures example

```
MODULE swapmod
  IMPLICIT NONE
  INTERFACE swap
    MODULE PROCEDURE swap_real, swap_char
  END INTERFACE
CONTAINS
  SUBROUTINE swap_real(a, b)
    REAL, INTENT(INOUT) :: a, b
    REAL :: temp
    temp = a; a = b; b = temp
  END SUBROUTINE
  SUBROUTINE swap_char(a, b)
    CHARACTER, INTENT(INOUT) :: a, b
    CHARACTER :: temp
    temp = a; a = b; b = temp
  END SUBROUTINE
END MODULE swapmod
```

```
PROGRAM switch
 USE swapmod
 IMPLICIT NONE
 CHARACTER :: n,s
 REAL :: x,y
 n = 'J'
 s = 'S'
 x=10
 y=20
 PRINT *,x,y
 PRINT *,n,s
 CALL swap(n,s)
 CALL swap(x,y)
 PRINT *,x,y
 PRINT *,n,s
END PROGRAM
```

Operator overloading

- To improve readability of Fortran source code it is possible to *overload* existing operators – or even create ones of your own
- Operator overloading applies to the mathematical operations like '+', '-', '*', '/' and assignment '='
  - Own operators defined with two dots, e.g., ".op."
- Binary operators involve two operands – residing in the left and right hand sides (LHS & RHS) of the operator
  - Unary operator (e.g. minus A) only has one operator (RHS)

Operator overloading

- A practical implementation of operator overloading usually involve creating a module file where all the necessary components will be placed
  - Derived data TYPE definition of the associated elements
  - Creation of appropriate INTERFACE blocks to enable compiler to map the references to the new operators
- Operator overloading can optionally be implemented to cover several (intrinsic) data types, or a mixture of types

Overloading summation operator ‘+’

```
module overload
     type vector_t
        real :: x, y, z
     end type vector_t
     interface operator(+)
        module procedure vector_add
     end interface
contains
    function add(v1, v2) result(v3)
      type(vector_t) :: v3
      type(vector_t), intent(in) ::&
        v1, v2
      v3%x = v1%x + v2%x
      v3%y = v1%y + v2%y
      v3%z = v1%z + v2%z
     end function add
end module overload
```

- All the necessary code fractions are now placed in a MODULE file
- … and referenced from the user source code, e.g. :

```
PROGRAM main
USE overload
TYPE(vector_t) :: x1, x2, out
x1 = vector_t(1,2,3)
x2 = vector_t(10,20,30)
out = x1 + x2
print *,'out=', out%x, out%y, out%z
END PROGRAM main

! The output is:
out= 11.00000 22.00000 33.00000
```

Outline of the third part

- Optional procedure arguments
- SAVEd variables
- Special procedure attributes
  - Recursive, Elemental, Pure

Optional procedure arguments

- Procedure arguments can be defined as OPTIONAL
  - Some predefined default value used for arguments not provided
- The presence of the optional arguments can be inquired with PRESENT clause

```
real function average(x, low, up)
 implicit none
 real, dimension(:), intent(in) :: x
 real, intent(in), optional :: low, up
 real :: a, b
 integer :: i, icount
 a = -huge(a)
 b = huge(b)
 if (present(low)) a = low
 if (present(up)) b = up
 average = 0.0
 icount = 0
 do i = 1, size(x)
   if (x(i) >= a .and. x(i) <= b) then
     average = average + x(i)
     icount = icount + 1
   end if
 end do
 average = average/icount
end function average
```

Counting an average of a set of real numbers – optionally numbers outside [low,up] can be omitted from the average. The function can be called with either 1, 2 or 3 argumets, but the set of numbers has to be provided.

SAVEd variables

- By default objects in procedures are dynamically allocated upon invocation
- Only saved variables keep their value from one call to the next
  - SAVE attribute

    ```
    REAL, SAVE :: a
    ```
  - Variables assigned with a value upon declaration are equal to SAVE attribute (C programmers should note this!)

    ```
    REAL :: a = 1.0
    ```

Special attributes for procedures: RECURSIVE

- Recursion means calling a procedure within itself
- Triggered via RECURSIVE keyword

```
recursive function factorial(n) result(fac)
  integer, intent(in) :: n
  integer :: fac
  if (n==0) then
    fac=1
  else
    fac=n*factorial(n-1)
  end if
end function factorial
```

Special attributes for procedures: PURE

- PURE keyword indicates that the function is free of side effects
  - Such as a change in value of an input argument or global variable
- Intrinsic functions are always pure
- No (external) I/O is allowed in PURE procedures
- Pure procedure must specify intents of its all arguments
- The motivation is efficiency: Enables more aggressive compiler optimization and parallelization with e.g. OpenMP

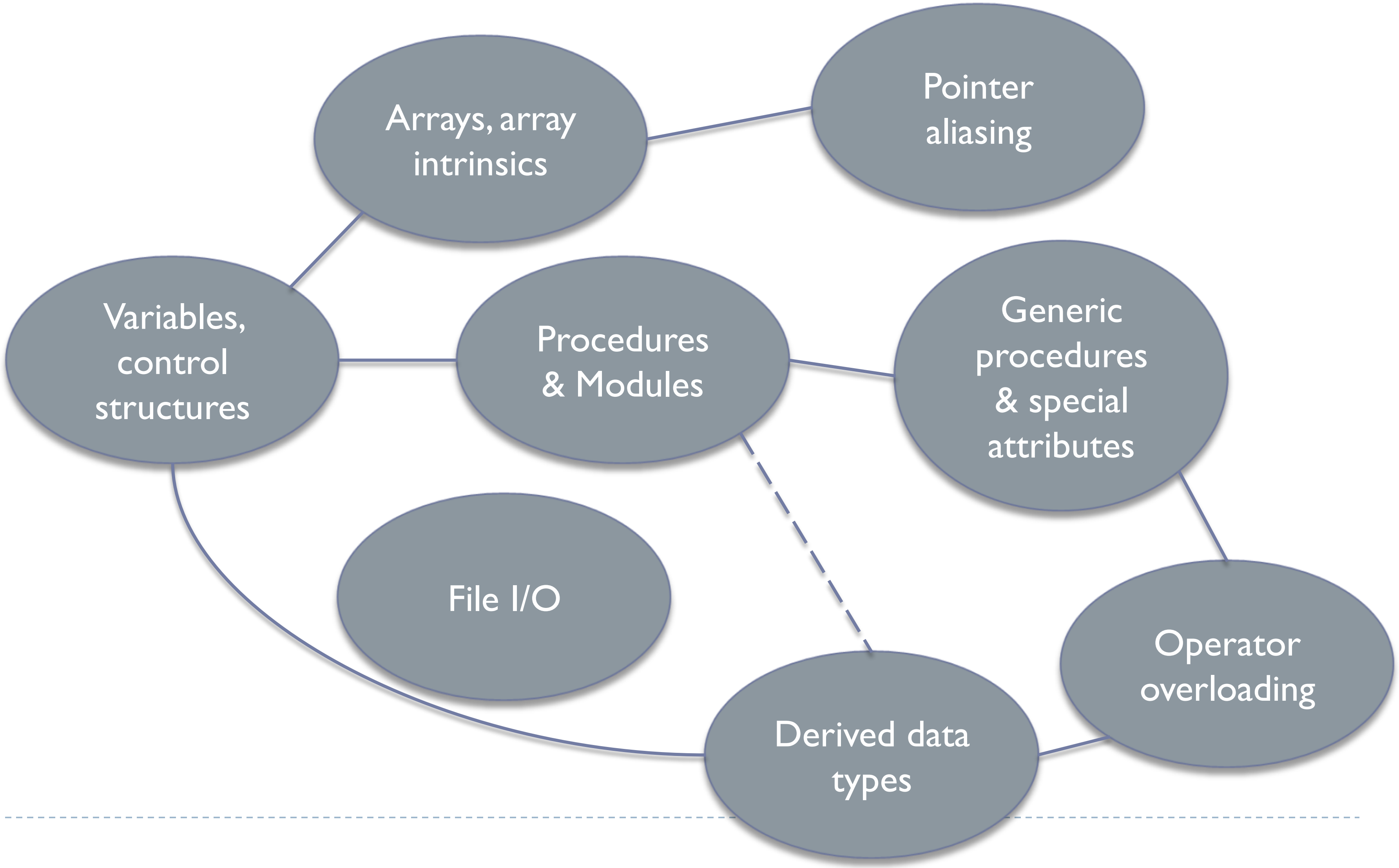
Special attributes for procedures: ELEMENTAL

- The ELEMENTAL attribute allows for declaring procedures that operate element-by-element
- The argument can be a scalar or an array of any shape
- This is another way for defining more general procedures

```
elemental real function f(x,y)
  real, intent(in) :: x, y
  f = sqrt(x**2 + y**2)
end function f

...
real, dimension(n,n) :: a, b, c
real, dimension(n) :: t, u, v
...
c = f(a,b)
v = f(t,u)
```

Fortran 95/2003 crash course summary



Programming assignment

The file vec.f95 contains a simple user interface for a program that performs basic linear algebra operations for vectors given by the user (here 3D for simplicity). Your task is to write the module containing the referred operations.

That is, you should include the definition of a suitable derived type. Then, implement operations for calculating the vector length (norm) and sum, substraction and dot and vector (cross) products between two vectors, and overload them to match with the syntax in vec.f95. Placeholders for these in vec_mod.f95.