Lecture VII: Object-oriented features
Outline

- Support for object-oriented style Fortran programming
  - Type parameters
  - Procedure pointers
  - Type extension
  - Polymorphism
- Fortran programming best practices
What is object-oriented programming?

- Program is separated into interacting objects
- Objects couple the data and the methods operating on the data
- Generic programming: the actual type of data and the associated implementation may be encapsulated and abstracted
- Motivation: Maintainability, readability and modifiability of the code are improved
- Fortran supports this approach by, e.g., generic procedures, type extension, polymorphic variables and type-bound procedures
Parameterisation of derived types

- Derived types can have type parameters

```fortran
    type matrix(prec, rows, cols)
      integer, kind :: prec
      integer, len :: rows, cols
      real(prec) :: mat(rows, cols)
    end type

    ! usage
    type(matrix(selected_real_kind(8), 10, 20) :: a
    type(matrix(selected_real_kind(4), n=n1, m=n2) :: b
```

- Type parameters have to be integers (and declared as such)
- They have to be either kind or len parameters and have the corresponding attributes
Parameterisation of derived types

- The type parameters can be given default values

```fortran
  type matrix(prec, rows, cols)
      integer, kind :: prec = selected_real_kind(8)
      integer, len :: rows=100, cols=100
      real(prec) :: mat(rows, cols)
  end type

  ! usage
  type(matrix(selected_real_kind(4), 10, 20)) :: a
  type(matrix) :: b ! double precision 100x100 matrix
```
Deferred type parameters

- A len type parameter is allowed to be a colon in type declaration of a pointer or allocatable entity

```fortran
character(len=:), pointer :: varchar
character(len=100), target :: name
character(len=200), target :: address
type(matrix(kind(1.0d0),:,:)), pointer :: A
real(kind=8), dimension(100,100), target :: B
...
varchar => name
...
varchar => address
...
A => B(1:50,1:50)
```
Abstract interfaces

- It is possible to define one interface for several (e.g. external) procedures having the same arguments but different names.

- The *abstract interface block* can be then used with the procedure statement to declare procedures:

```fortran
abstract interface
    subroutine subroutine_with_no_args
    end subroutine subroutine_with_no_args
    real function r_to_r(a,b)
        real, intent(in) :: a, b
    end function r_to_r
end interface
```

...  

```fortran
procedure(subroutine_with_no_args) :: sub1, sub2
procedure(r_to_r) :: xyz
! procedure statement can be used with explicit interface
procedure(func) :: func2
```
Pointers to procedures

- Instead of a data object, a pointer can be associated with a procedure
- A procedure pointer is declared by specifying that it is both a procedure and a pointer

```fortran
pointer :: sp
interface
  subroutine sp(a,b)
    integer, intent(in) :: a
    integer, intent(out) :: b
  end subroutine sp
end interface
! could be used like
sp => sub
call sp(a,b) ! calls sub(a,b)
```
Procedure pointer components

- A derived type may contain a procedure pointer
- E.g. define a type for representing a list of procedures (with the same interface)

```fortran
type process_list
  procedure(process_interface), pointer :: process
  type (process_list), pointer :: next
end type process_list
abstract interface
  subroutine process_interface(...)
    ...
  end subroutine process_interface
end interface
...
type(process_list) :: y(10)
call y(i)%process(...) ! invoke directly a list process
p => y(j)%process ! or assign a procedure pointer to one comp.
```
Type extension

- Creates new derived types by extending existing ones
- The new type inherits all the components and may add new ones

```plaintext
type person
  integer :: id
  character(len=10) :: name
  integer :: age
end type

! a new type can be formed as an extension as
type, extends(person) :: employee
  character(len=11) :: social_security_id
  real :: salary
end type
```
Type extension

- The new type inherits also all type parameters. New type parameters can be added.

```fortran
  type matrix(prec, rows, cols)
    integer, kind :: prec
    integer, len :: rows, cols
    real(prec) :: mat(rows, cols)
  end type

  type, extends(matrix) :: labelled_matrix(max_label_length)
    integer, len :: max_label_length
    character(max_label_length) :: label = ''
  end type labelled_matrix

  ... type labelled_matrix(kind(0.0),10,20,200)) :: x
```
Polymorphism

- The data type of a *polymorphic variable* may vary at run time
- It has to be a pointer or allocatable, and it is declared with the *class* keyword:

  ```
  type point
    real :: x, y
  end type
  class(point), pointer :: p
  ```

- The type named in the class attribute must be an extensible derived type
Polymorphism

Now for example

```plaintext
real function distance(a, b)
    class(point) :: a, b
    distance = sqrt((a%x-b%x)**2 + (a%x-b%x)**2)
end function distance

! this can take arguments that are of type point but also
! any extension of it, e.g.
type, extends(point) :: data_point
    real, allocatable :: data(:)
end type
```

A polymorphic variable can be either an array or a scalar

- In an polymorphic array all elements must be of same type
Polymorphism

- Unlimited polymorphic pointer may refer to objects of any type
- The value of an unlimited polymorphic pointer cannot be accessed directly, but the object as a whole can be used
  - e.g. passed as an argument

```
class(*), pointer :: univp
type(triplet), pointer :: tripp
real, pointer :: realp
...
univp => tripp ! valid
univp => realp ! valid
tripp => univp ! valid if dynamic type matches
realp => univp ! invalid
```
Polymorphism

To execute alternative code depending on the dynamic type of a polymorphic entity, the select type construct is used:

class(particle) :: p

...  
print *, p%position, p%velocity, p%mass
select type(p)
type is (charged_particle)
  print *, 'Charge: ', p%charge
class is (charged_particle)
  print *, 'Charge: ', p%charge
  ! may have other attributes
type is (particle)
  ! nothing extra
class default
  print *, 'may have other unknown attributes'
end select
Type-bound procedures

- These are procedures which are invoked through an object, and the actual procedure executed depends on the dynamic type of the object.
- Corresponds to a "method" of true OOP languages.

```fortran
module mod_mytype
  type mytype
    private
    real :: myvalue(3) = 0.0
  contains
    procedure :: write => write_mt
    procedure :: reset
  end type mytype
  private :: write_mt, reset
  contains
    subroutine write_mt(this,unit)
      class(mytype) :: this
      integer, optional :: unit
      if (present(unit)) then
        write(unit,*) this%myvalue
      else
        write(*,*) this%myvalue
      end if
    end subroutine write_mt
  end subroutine write_mt
  ...
```
Type-bound procedures

- Each type-bound procedure declaration specifies the name of the binding, and the name of the actual procedure.

- The type-bound procedures are invoked as component procedure pointers of the object.

  - For example, the procedures of the last example would be invoked as:
    
    ```
    call x%write(10)
    call x%reset
    ! these are equivalent to
    ! call write_mt(x,10) or call reset(x), but being
    ! private they are only accessible through the object
    ! outside the module mod_mytype
    ```
Best practices - general considerations

- **Clarity** first - if the program source code is easy to read for you, it will be that also for the next contributor, as well as the compiler
  - Comment and document your code
- **Write structured, simple code**
  - Employ modules, write short and simple procedures
- Express what you want to express *simply, concisely and clearly*; avoid gimmicks & hacks
- Do not re-invent the wheel - *use libraries* and reuse code elsewhere
Best practices - readability

- Write standard-compliant, readable, portable code that is easy to modify
  - Isolate machine/compiler-dependent solutions with preprocessor pragmas and document them
- Document your code (write a readme file and distribute it with the source code)
  - How to compile, run and how to interpret results
- Don’t spare in comments
  - Not only describing what’s happening but also why
- Use self-explaining variable and procedure names
  - compare ”A” vs ”coefficient_matrix”
Best practices - syntax

- Use modern control structures and avoid obsolete ones:
  - do ... end do
  - select case ... case ... end select
  - if ... else if ... end if
  - where... elsewhere ... end where
  - forall... end forall

- Do not enumerate lines (old F77 practice)

- Employ array syntax and other array features
Best practices - variables

- Define all variables
  - always have "implicit none"
  - define constants as parameters
- Avoid global variables; expose data as little as possible by minding private and public attributes
- Encapsulate conceptually related variables into derived types
- Initialize variables
  - An uninitialized variable is not necessary zero!
Best practices - procedures

- Each procedure should do *one* thing and do it well
  - The implementation details and local data should be hidden from the caller
- Put procedures into modules - closely related procedures to the same module
- Define *interfaces* for external procedures if you have to use them
- Define intent(in|inout|out) for all procedure arguments
Best practices - input/output

- Implement as simple user input interface if possible
  - employ command-line arguments
  - allow free formatting in input files
- Let the user control the output verbosity level
- Implement sanity checks for user input, possibly recovering from insensible input
  - Have default values for all input parameters if feasible
- Put all I/O into separate procedures (into a same module)
  - Spreading I/O everywhere into program code hinders compiler optimization
- I/O is a typical performance bottleneck!
  - Use binary data and stream (perhaps asynchronous) I/O for other than log files
Best practices - debugging

- Generate a set of tests for your code and run the set frequently
  - When adding new features, just add more tests
- Use debuggers (e.g. gdb) and compiler features (e.g. -fbounds-check) to bug catching
Best practices - performance

- First do it correctly, and only then more efficiently (and still correctly)
  - “Premature code optimization is the root of all evil”
  - Chosen algorithm defines the majority of program performance
- 90/10 rule - typically ~90% of the execution time is being spent on ~10% of the source code lines
  - Identify these parts by profiling and focus all optimization efforts into those
The End

- We have now covered the most useful features of the Fortran (2008) programming language
- Scientific software development in a nutshell:
  - Employ the best algorithm
  - Write
    - standard-compliant
    - clear & concise
    - modular & structured
    - commented
  
  code
Programming assignment

Revisit the vector algebra program (Assignment #4) and generalize it to treat arbitrary-sized vectors (you can omit the cross product, since its generalization is non-trivial).

Let’s do this by rewriting the vector_algebra module to employ (some) the presented object-oriented features, e.g.

- The vector type has the operations type-bound
- The type is being parameterized for precision
- The procedures have polymorphic arguments