Object oriented programming with Fortran 200X

Alberto F. Martín Javier Principe
{amartin,principe}@cimne.upc.edu

Large Scale Scientific Computing group
International Center for Numerical Methods in Engineering (CIMNE)

Universitat Politècnica de Catalunya

July 26, 2017

Remark

The PDF of the slides, the codes of the accompanying exercices and examples, together with scripts for automatic compilation, are available at a public Git repository that can be cloned as:

Execute me on a Linux terminal

```
$ git clone https://gitlab.com/femparadmin/fmw_oop_bcn_material.git
```

Content

Outline

The LSSC group at CIMNE/UPC

Members:

- Santiago Badia (Group leader, Professor, UPC)
- Jesus Bonilla (PhD student, UPC)
- Alberto F. Martín (Senior researcher, CIMNE)
- Eric Neiva (PhD student, UPC)
- Marc Olm (PhD student, UPC)
- Javier Principe (Associate Professor, UPC)
- Francesc Verdugo (JdC researcher, CIMNE)

We develop `FEMPAR`: F200X OO massively parallel multilevel FEM framework

FEMPAR goals

- It is an open source scientific software library for the high-performance scalable simulation of complex multiphysics problems described by partial differential equations.
- It provides a rich set of algorithms for the discretization step:
  - Arbitrary-order finite element methods for H^1, $H(\mathbf{curl})$, and $H(\mathrm{div})$ spaces
  - Discontinuous Galerkin methods
  - B-spline discretizations
  - Unfitted finite element techniques on cut cells, combined with *h*-adaptivity
- A bulk-asynchronous multilevel FE framework which can be customized for the problem at hand. In particular, it provides massively parallel linear solvers.

FEMPAR

Member • Member • Member • Member • Member • Member • Member • Member • Member • Member • Member • Member

HiQ

2014

Click on us!

FEMPAR history

- 01/2011: Launched in as an in-house code by S. Badia, A. F. Martín, and J. Principe, at CIMNE/UPC
- 01/2011-03/2015: worked hard in the algorithms...using structured programming
- 11/2014: FEMPAR attained perfect weak scalability for up to 458,672 cores on JUQUEEN (Germany), the largest supercomputer in Europe (at that time), solving up to 60 billion unknowns
- 03/2015: Developing was out of control ...
- 03/2015-06/2017: working hard in object oriented redesign
- 05/2017: released under GNU/GPLv3

Outline

Why object oriented programming?

- Reduction of programming effort (essential for large software projects)
- Permits easy code reuse (between projects)
- Abstraction is essential...
- ...as it makes the software a closer expression of your algorithm

Why object oriented programming?

- Reduction of programming effort (essential for large software projects)
- Permits easy code reuse (between projects)
- Abstraction is essential...
- ...as it makes the software a closer expression of your algorithm

```
real :: A(10,10),B(10,10),C(10,10)
type(matrix) :: D,L,U
! F77 matrix multiplication
do i=1,10
   do j=1,10
      do k=1,10
         C(j,i) = A(i,k) * B(k,j)
      end do
   end do
end do
...
! F90 alternative
C = matmul(A,B)
...
! F03 + OO programming
D = L * U
```

```
type :: matrix
   real :: values(10,10)
 contains
   procedure :: product
   generic   :: operator(*) => product
end type matrix

function product(A,B) result(C)
  type(matrix) :: A,B,C
  do i=1,10
     do j=1,10
        do k=1,10
           C%values(j,i) = A%values(i,k) *
                  B%values(k,j)
        end do
     end do
  end do
end function product
```

Why object oriented programming?

- Reduction of programming effort (essential for large software projects)
- Permits easy code reuse (between projects)
- Abstraction is essential...
- ...as it makes the software a closer expression of your algorithm

The OO implementation of the CG algorithm:

```
1  r = b - A*x
2  p = r
3  do while(.not.converged)
4  alpha = (r*r)/((A*p)*p)
5  x     = x + alpha * p
6  rnew  = r - alpha * A*p
7  beta  = (rnew*rnew)/(r*r)
8  p     = r + beta * p
9  converged = ( sqrt(rnew*rnew) < tol)
10 end do
```

A snapshot from Saad's book:

ALGORITHM 6.17: Conjugate Gradient

1. *Compute* $r_0 := b - Ax_0$, $p_0 := r_0$.
2. *For* $j = 0, 1, \ldots,$ *until convergence Do:*
3. $\quad \alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4. $\quad x_{j+1} := x_j + \alpha_j p_j$
5. $\quad r_{j+1} := r_j - \alpha_j A p_j$
6. $\quad \beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7. $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
8. *EndDo*

Weakness of structured programming

Following R.C. Martin (uncle Bob)[1], there are four main symptoms of bad software design. They are not independent but closely related concepts:

- Rigidity
- Fragility
- Immobility
- Viscosity

[1] https://sites.google.com/site/unclebobconsultingllc/

Rigidity

- The software is difficult to change, even in simple ways.
- A small change triggers a number of subsequent changes because of **dependencies** to other modules
- Even minor tasks become endless
- Resistance to fix non-critical problems
- Addition of ad-hoc patches that increase rigidity and fragility
- Programmers are always thinking: "I thought it was easier!!!!"

Fragility

- Is the tendency of the software to break in many places when a change is introduced
- Often the break occurs at places without conceptual relationship with the area where the change was introduced
- Therefore, fixing is difficult because introducing a fix triggers many other problems
- Debugging time increases dramatically
- The software becomes difficult to maintain
- Programmers are reluctant to make changes
- The concept is closely related to rigidity and, again, is due to uncontrolled **dependencies**

Immobility

- Is the inability to **reuse** code from other project or even between parts of the same project
- The code is so tangled that it is impossible to isolate modules. Once again, this is because there are many **dependencies**
- The effort and the risk of separating a module from the project is so high that it is easier to code the algorithm again

Viscosity

- There are two forms of viscosity: viscosity of the design and viscosity of the environment
- Viscosity of the design:
  - It is not clear how to make changes
  - There are many (more than one) ways of doing it
  - The easier (hacks) make the software worst (more rigid, fragile,...) and the good ones are difficult
- Viscosity of the environment:
  - The development environment is slow and inefficient
  - E.g. compile times are very long

Example: using global variables

A rigid, fragile, immobile program to integrate

$$\frac{dy}{dt} = -\alpha(t)y^2 \quad \text{with} \quad y(0) = y0$$

in the interval $[0, T]$ using the Euler method (dividing the interval in n steps of size dt)

$$y_{n+1} = y_n - \alpha_n \, dt \, y_n^2$$

where $\alpha_n = \alpha(t_n)$ and $t_n = ndt$.

Example: using global variables

```
1  module properties
2    real :: alpha
3  end module properties
```

```
1  ! Integrate dy/dt = -alpha(t) y^2 with y(0)=y0
2  program globals
3    use properties
4    use problem
5    implicit none
6    integer :: n
7    real    :: yold, ynew, t, dt
8    yold  = 1.0
9    dt = 0.1
10   t  = 0.0
11   n  = 100
12   do i=1,n
13      t = t + dt
14      call update(t)
15      ynew = yold - alpha * dt *yold**2
16      call output(ynew)
17   end do
18 end program globals
```

```
1  module problem
2    use properties
3  contains
4    subroutine update(t)
5      alpha = 1.0 + exp(-t)
6      ! alpha = alpha + 1/t
7    end subroutine update
8  end module problem
```

```
1  module output
2    use properties
3  contains
4    subroutine print(y)
5      implicit none
6      real :: y
7      write(*,*) y, -alpha*y**2
8    end subroutine print
9  end module output
```

Exercise: solution of linear systems

We aim to solve a system of equations $Ax = b$.

There are two families of methods:

- Direct methods, which have two phases:
  1. Factorize the matrix as $A = LU$ where L (U) is a lower (upper) triangular matrix
  2. Solve the system in two steps, $Ly = b$ (forward substitution) and $Ux = y$ (backward substitution). It can be executed for different values of b
- Iterative methods: starting from an initial estimation x_0, iteratively correct x_i using the residual $Ax_i - b$. They also have two phases:
  1. Setup (allocate) temporary data structures required by the algorithms
  2. Apply to algorithm to a given right hand side b

Exercise: solution of linear systems

The entries of the matrix A can be stored in different ways in memory, which requires different data structures

- full storage (all the entries are stored)
- band storage
- compressed sparse storage

Question

Do we need to implement each solution algorithm (direct or iterative) for each storage type?

We will work with a refactorization of a legacy code[2] that solves the linear system involving the so-called Wathen matrix (that e.g., can arise from a FEM discretization)

[2] https://people.sc.fsu.edu/~jburkardt/f_src/wathen/wathen.html

Outline

Encapsulation

- OO programming requires encapsulation: data and functions operating with them are bundled
- Fortran 2003 (F2003) added the capability for a **derived data type** to encapsulate procedures, thus making it an **object**

```
1  subroutine agenda(persons)
2   use person_mod
3   type(person_t), intent(in) :: persons(:)
4   ...
5   if( get_age(person(i)) >= &
6     & get_age(person(j)) ) then
7       ...
8   end if
9   if(person(i)%age > person(j)%age ) then
10      ...
11  end if
12  ...
13 end subroutine agenda
```

```
1  module person_mod
2    type person_t
3       integer       :: age
4       logical       :: gender
5    end type person_t
6  contains
7    function get_age(this)
8      type(person_t), intent(in) :: this
9      integer :: get_age
10     get_age = this%age
11   end function get_age
12   ...! Rest of TBPs
13 end module person_mod
```

```
1  subroutine agenda(persons)
2   type(person_t), intent(in) :: persons(:)
3   ...
4   if(persons(i)%get_age() > person(j)%get_age()) then
5      ...
6   end if
7   ...
8  end subroutine agenda
```

```
1  module person_mod
2    type person_t
3       integer       :: age
4       logical       :: gender
5     contains
6       procedure :: get_age
7       ...
8    end type person_t
9  contains
10   function get_age(this)
11     class(person_t), intent(in) :: this
12     integer :: get_age
13     get_age = this%age
14   end function get_age
15   ...! Rest of TBPs
16 end module person_mod
```

Type-Bound procedures

- F2003 added a `contains` keyword to its derived types to separate type's data definitions from its procedures
- Anything that appears after the `contains` keyword in a derived type must be a type-bound procedure declaration

General syntax of type-bound procedure declaration

```
PROCEDURE [(interface-name)] [[,binding-attr-list ]::] binding-name[=> procedure-name]
```

- Anything in brackets is optional in the TBP syntax. Thus, at minimum, a TBP is declared with the `procedure` keyword + `binding-name`, i.e., the TBP name
- Option `interface-name` and `binding-attr-list` to be covered later on
- `procedure-name` is the name of the procedure that implements the TBP. This option is required only if `procedure-name` differs from `binding-name`
- `procedure-name` can be either a **module procedure** or an **external procedure with explicit interface** (see next slide)

Type-Bound procedures (continued)

```
1  module person_mod
2    type person_t
3       integer    :: age
4       logical    :: gender
5    contains
6       procedure :: initialize => person_init
7       procedure :: get_age
8    end type person_t
9
10 contains
11
12   subroutine person_init(this, age, gender)
13     implicit none
14     class(person_t)   , intent(inout) :: this
15     integer           , intent(in)    :: age
16     integer           , intent(in)    :: gender
17     this%age    = age
18     this%gender = gender
19   end subroutine person_init
20
21   function get_age(this)
22     class(person_t), intent(in) :: this
23     integer :: get_age
24     get_age = this%age
25   end function get_age
26
27 end module person_mod
```

person_initialize as a module procedure

```
1  module person_mod
2   type person_t
3    ... ! type components (see left snippet)
4   end type person_t
5
6   interface
7    subroutine person_init(this, age, gender)
8     import :: person_t
9     implicit none
10    class(person_t)   , intent(inout) :: this
11    integer           , intent(in)    :: age
12    integer           , intent(in)    :: gender
13  end subroutine person_init
14  end interface
15 end module person_mod
```

```
1  ! External procedure. It may go into separate
2  ! source file to reduce build times
3  subroutine person_init(this, age, gender)
4     use person_mod, only : person_t
5     implicit none
6     class(person_t)   , intent(inout) :: this
7     integer           , intent(in)    :: age
8     integer           , intent(in)    :: gender
9      this%age    = age
10     this%gender = gender
11   end subroutine person_init
```

person_initialize as an external procedure

Type-Bound procedures (continued)

- Using the modules in the previous slide, the `initialize` TBP is invoked as:

```
1 use person_mod
2 type(person_t) :: person          ! Declare an instance of person_t
3 call person%initialize(35, MALE) ! Invoke initialize TBP to initialize person
```

- Syntax for invoking a TBP is very similar to accessing a data component
- Name of the component is preceded by the instance name separated by % sign
- In our example, component name is `initialize`, and variable name `person`
- Thus, we type `person%initialize(args)` to access the `initialize` TBP
- The above example calls the `initialize` subroutine, and passes 35 for *age*, `MALE` for *gender* (see previous slide)

Type-Bound procedures (continued)

- Using the modules in the previous slide, the `initialize` TBP is invoked as:

```
1 use person_mod
2 type(person_t) :: person          ! Declare an instance of person_t
3 call person%initialize(35, MALE) ! Invoke initialize TBP to initialize person
```

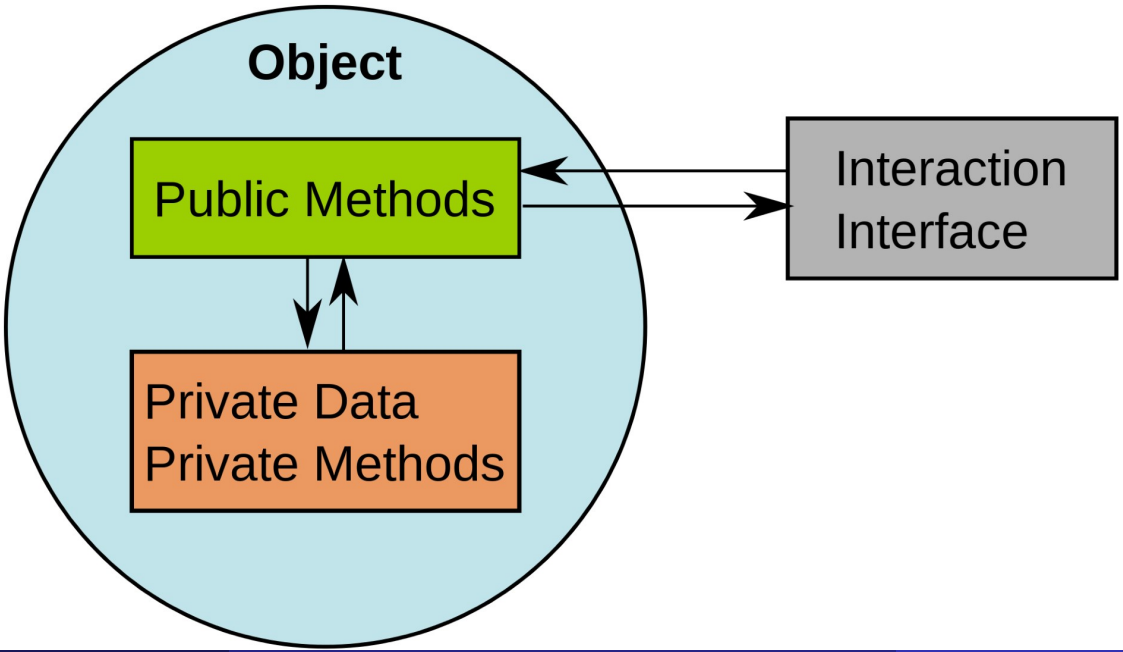
- What about the first dummy argument, `this`, in `initialize` subroutine? It is known as the **passed-object dummy argument**
- **By default**, the passed-object dummy argument MUST BE the first dummy argument in the subroutine implementing the TBP (this can be changed using the pass or nopass options)
- It receives as actual argument the instance on which the TBP was invoked, `person` in our example
- The passed-object argument MUST BE declared `class` (i.e., polymorphic as discussed later)
- **Coding style suggestion**: always use a reserved name for passed-object dummy arguments (e.g., `this` or `self`) to distinguish them from regular dummy arguments of the subroutine implementing TBP

Data hiding

- Information hiding is a technique in OOP that allows to expose an object and its procedures as a "black box"
- That is, the programmer can use an object without any knowledge on how its data components are laid out, nor how its procedures are implemented
- Inquiry functions like the `initialize` TBP in the previous slides, are common with information hiding
- Inquiry functions let the object's implementer change its internal organization without affecting the programs that use the object

Data hiding

- Information hiding is a technique in OOP that allows to expose an object and its procedures as a "black box"
- That is, the programmer can use an object without any knowledge on how its data components are laid out, nor how its procedures are implemented
- Inquiry functions like the `initialize` TBP in the previous slides, are common with information hiding
- Inquiry functions let the object's implementer change its internal organization without affecting the programs that use the object



Data hiding

- Information hiding is a technique in OOP that allows to expose an object and its procedures as a "black box"
- That is, the programmer can use an object without any knowledge on how its data components are laid out, nor how its procedures are implemented
- Inquiry functions like the `initialize` TBP in the previous slides, are common with information hiding
- Inquiry functions let the object's implementer change its internal organization without affecting the programs that use the object

Dependencies

Clients depend on the type *only* through the interface of the function

Therefore the design of stable interfaces, that are suitable for the user needs, and changing requirements over time, becomes essential.

Information hiding with F2003

- To **enable** information hiding, F2003 provides a `private` keyword
- To **disable** it, a `public` keyword
- Both `private` and `public` can be placed on derived type data and TBP components, and on module entities (variables, types, procedures, ...)
- **By default**, all derived type data and TBP components, and module entities are declared `public`

Information hiding with F2003

```
1  use person_mod
2  type(person_t)      :: person
3  call person%initialize(35, MALE)
4  if(person%age > 18) then
5  ! you get a compilation error
```

```
1  module person_mod
2    implicit none
3
4    ! Hide module data types and procedures
5    private
6
7    ! Publicate data types and module procedures
8    public :: person_t
9
10   type person_t
11    private     ! Hide data components of person_t
12    integer    :: age
13    integer    :: gender
14   contains
15    private     ! Hide by default TBPs of person_t
16    procedure, public :: initialize => person_init
17    procedure, public :: get_age
18   end type person_t
19
20 contains
21
22   ... ! Private TBPs implementation
23
24 end module person_mod
```

- This code uses information hiding both in the host module and in person_t
- The private statement located at the top of the module enables information hiding on all module entities
- For example, the get_age module procedure (do not confuse with the get_age TBP) is hidden as a result of this private statement

Information hiding with F2003

```
1  use person_mod
2  type(person_t)      :: person
3  call person%initialize(35, MALE)
4  if(person%age > 18) then
5  ! you get a compilation error
```

```
1  module person_mod
2    implicit none
3
4    ! Hide module data types and procedures
5    private
6
7    ! Publicate data types and module procedures
8    public :: person_t
9
10   type person_t
11    private     ! Hide data components of person_t
12    integer    :: age
13    integer    :: gender
14   contains
15    private     ! Hide by default TBPs of person_t
16    procedure, public :: initialize => person_init
17    procedure, public :: get_age
18   end type person_t
19
20 contains
21
22   ... ! Private TBPs implementation
23
24 end module person_mod
```

- We added the `private` statement on the data components of `person_t`
- Thus, e.g., the only way the module's user can obtain the value of the age data component is thorough the `get_age` TBP (which is declared `public`)
- Note also the `private` statement right after the start of the `contains` keyword in type `person_t`
- If u want your TBPs to be also private, this second statement is absolutely necessary (otherwise TBPs are `public` by default)
- `private` TBPs (and data components) are only accessible within the host module!

Type constructors

```
1  module person_mod
2    implicit none
3
4    ! Hide module data types and procedures
5    private
6
7    ! Publicate data types and module procedures
8    public :: person_t, constructor
9
10   type person_t
11    private    ! Hide data components of person_t
12    integer    :: age
13    integer    :: gender
14   contains
15    private    ! Hide by default TBPs of person_t
16    procedure          :: person_initialize
17    procedure, public  :: get_age
18   end type person_t
19
20 contains
21
22   function constructor(age,gender)
23     integer, intent(in) :: age
24     integer, intent(in) :: gender
25     type(person_t)      :: constructor
26     call constructor%person_init(age,gender)
27   end function constructor
28
29   ... ! get_age() and person_initialize as
           before
30
31 end module person_mod
```

```
1 use person_mod
2 type(person_t)     :: person
3 person = constructor(35, MALE)
```

- The subroutine `person_initialize` is an example of a type constructor, i.e. a procedure that receives the required information and fills the structure
- However, it is more common to use function *returning* the type
- Observe that access to `person_initialize` is allowed as the call is performed within the host module
- The problem here is that `constructor` may well be defined somewhere else in the host program (name collision).

Type Overloading

```
1  module person_mod
2    implicit none
3
4    ! Hide module data types and procedures
5    private
6
7    ! Publicate data types
8    public :: person_t
9
10   type person_t
11    private    ! Hide data components
12    integer   :: age
13    integer   :: gender
14   contains
15    private    ! Hide TBPs by default
16    procedure         :: person_init ! private
17    procedure, public :: get_age
18   end type person_t
19
20   ! Overload the generic interface person_t
21   ! with the  constructor function
22   ! (which is now private!!!)
23   interface person_t
24     module procedure constructor
25   end interface person_t
26
27 contains
28   ... ! get_age(),  person_init and constructor
29       ! as before
30 end module person_mod
```

```
1 program person_program
2   use person_mod
3   type(person_t) :: person
4   ! Invoke constructor through the
5   ! person_t generic interface
6   person = person_t(35, MALE)
7 end program person_program
```

- The F03 standard provides a standard name to refer to constructors while still bypassing this name collision issue
- It allows to overload the name of a derived type with a generic interface
- The generic interface acts as a wrapper for our `constructor` function
- Our `constructor` function is now `private` and can be invoked thorough the `person_t` generic interface (which is `public` automatically after publicating the name of the data type)

Outline

Inheritance in F2003

```
1  type person_t
2     integer :: age
3     integer :: gender
4  end type person_t
5
6  type, extends(person_t) :: employee_t
7    real      :: salary
8  end type employee_t
9
10 type, extends(employee_t) :: manager_t
11    real     :: bonus
12 end type manager_t
```

- We have a `manager_t` type that inherits components from `employee_t`, which in turn inherits components from `person_t`
- Programmer indicates the inheritance relationship with the `extends` keyword + name of parent type in parentheses
- A type that extends another is known as a type extension (e.g., `employee_t` is a type extension of `person_t`)
- A type without any `extends` keyword is known as a base type (e.g., `person_t`)

Inheritance in F2003

```
1  type person_t
2     integer :: age
3     integer :: gender
4  end type person_t
5
6  type, extends(person_t) :: employee_t
7    real      :: salary
8  end type employee_t
9
10 type, extends(employee_t) :: manager_t
11    real     :: bonus
12 end type manager_t
```

- A type extension inherits all components of its parent (and ancestors)
- A type extension can define additional components (but not necessarily)
- `employee_t` defines `salary` as additional component, inherits `age`/`gender` from `person_t`
- `manager_t` defines `bonus` as additional component, inherits all components from `employee_t` and `person_t`

Inheritance in F2003

```
1  type person_t
2     integer :: age
3     integer :: gender
4  end type person_t
5
6  type, extends(person_t) :: employee_t
7    real      :: salary
8  end type employee_t
9
10 type, extends(employee_t) :: manager_t
11    real     :: bonus
12 end type manager_t
```

- Multiple ways we may use to access to the age component of `manager`
- A type extension includes an **implicit** component with same name/type as parent
- Helpful to operate on components specific to a parent type, also to illustrate an important relation among child and parent types
- In OOP we say that the child and parent types have a "*is a*" relationship (e.g., "*a `employee_t` is a `person_t`*", etc.)
- "is a" does NOT imply the converse (e.g., "*a `person_t` is NOT a `employee_t`*", the latter indeed has components not found in `person_t`)

```
1  type(manager_t) :: manager       ! declare manager as an instance (object) of static type manager_t
2  ...
3  manager%age                      ! access age component of manager directly
4  manager%employee_t%age           ! access age component of manager thorough parent class
5  manager%employee_t%person_t%age  ! access age component of manager thorough path from leaf to root
```

Inheritance and Type-Bound procedures

- Recall that a child type inherits all components from parent/ancestor types
- This applies to both data and procedures in the case of F2003 derived types
- In the code below, `employee_t`+`manager_t` inherit `initialize` from `person_t`
- We can thus call `get_age` with either a `person_t`/`employee_t`/ `manager_t` instance (subroutines implementing TBPs are **polymorphic**, next topic)

```
1  type person_t
2    integer    :: age
3    integer    :: gender
4  contains
5    procedure :: initialize
6  end type person_t
7
8  type, extends(person_t) :: employee_t
9    real       :: salary
10 end type employee_t
11
12 type, extends(employee_t) :: manager_t
13   real       :: bonus
14 end type manager_t
```

```
1  use person_mod
2  type(person_t)   :: person
3  type(employee_t) :: employee
4  type(manager_t)  :: manager
5  integer          :: age
6
7  age = person%get_age()
8
9  age = employee%get_age()
10
11 age = manager%get_age()
```

Procedure overriding

```
1  module person_mod
2    type person_t
3       integer    :: age
4       integer    :: gender
5     contains
6       procedure :: income => person_income
7    end type person_t
8    type, extends(person_t) :: employee_t
9       real       :: salary
10    contains
11      procedure :: income => employee_income
12   end type employee_t
13   type, extends(employee_t) :: manager_t
14      real       :: bonus
15    contains
16      procedure :: income => manager_income
17   end type manager_t
18 contains
19   function person_income(this)
20     class(person_t), intent(in) :: this
21     real :: person_income
22     person_income = 0.0
23   end function person_income
24   function employee_income(this)
25     class(employee_t), intent(in) :: this
26     real :: employee_income
27     employee_income = this%salary
28   end function employee_income
29   function manager_income(this)
30     class(manager_t), intent(in) :: this
31     real :: manager_income
32     manager_income = this%salary + this%bouns
33   end function manager_income
34 end module person_mod
```

```
1 use person_mod
2 type(person_t)   :: person
3 type(employee_t) :: employee
4 type(manager_t)  :: manager
5 real             :: icnome
6
7 income = person%income()
8 income = employee%income()
9 income = manager%income()
```

- An `income` TBP is declared for `person_t` and `employee_t`, and `manager_t`
- The one of `employee_t` overrides the counterpart in `person_t` and the one in `manager_t` overrides the counterpart in `employee_t`
- Therefore, `call employee%income(...)` actually calls `employee_income` and `call manager%income(...)` actually calls `manager_income`

Procedure overriding

```
1  module person_mod
2    type person_t
3       integer    :: age
4       integer    :: gender
5     contains
6       procedure :: income => person_income
7    end type person_t
8    type, extends(person_t) :: employee_t
9       real       :: salary
10    contains
11      procedure :: income => employee_income
12   end type employee_t
13   type, extends(employee_t) :: manager_t
14      real       :: bonus
15    contains
16      procedure :: income => manager_income
17   end type manager_t
18 contains
19   function person_income(this)
20     class(person_t), intent(in) :: this
21     real :: person_income
22     person_income = 0.0
23   end function person_income
24   function employee_income(this)
25     class(employee_t), intent(in) :: this
26     real :: employee_income
27     employee_income = this%salary
28   end function employee_income
29   function manager_income(this)
30     class(manager_t), intent(in) :: this
31     real :: manager_income
32     manager_income = this%salary + this%bouns
33   end function manager_income
34 end module person_mod
```

```
1 use person_mod
2 type(person_t)   :: person
3 type(employee_t) :: employee
4 type(manager_t)  :: manager
5 real             :: icnome
6
7 income = person%income()
8 income = employee%income()
9 income = manager%income()
```

- Passed-object dummy argument is declared as `person_t` in `person_income`, `employee_t` in `employee_income` and `manager_t` in `manager_income`
- Recall that procedure's passed-object argument data type must match the one of the type that defined it!
- The rest of dummy arguments of the overriding TBP MUST BE identical (same name, type, number, etc.) to the ones of the overridden one
- This is because it must be possible to invoke both TBPs in the same manner

Procedure overriding

```
1  module person_mod
2    type person_t
3       integer    :: age
4       integer    :: gender
5     contains
6       procedure :: income => person_income
7    end type person_t
8    type, extends(person_t) :: employee_t
9       real       :: salary
10    contains
11      procedure :: income => employee_income
12   end type employee_t
13   type, extends(employee_t) :: manager_t
14      real       :: bonus
15    contains
16      procedure :: income => manager_income
17   end type manager_t
18 contains
19   function person_income(this)
20     class(person_t), intent(in) :: this
21     real :: person_income
22     person_income = 0.0
23   end function person_income
24   function employee_income(this)
25     class(employee_t), intent(in) :: this
26     real :: employee_income
27     employee_income = this%salary
28   end function employee_income
29   function manager_income(this)
30     class(manager_t), intent(in) :: this
31     real :: manager_income
32     manager_income = this%salary + this%bouns
33   end function manager_income
34 end module person_mod
```

```
1  use person_mod
2  type(person_t)   :: person
3  type(employee_t) :: employee
4  type(manager_t)  :: manager
5  real             :: icnome
6
7  income = person%income()
8  income = employee%income()
9  ! How the income of a manager would as an employee?
10 income = manager%employee_t%income()
```

- It is still possible to invoke the version of a TBP defined by a parent type
- Recall that each type extension has an implicit data component with name/type equivalent to the one of the parent type
- We can use this data component to access the (non-overridden) TBP version of the parent (see code snippets left and above)

Non-overridable procedures

- Sometimes we may not want a child to override a parent's TBP
- We can use the `non_overridable` binding-attribute for such purpose
- Transforms **dynamic binding** into **static binding**
- This hint might be used by compiler to introduce optimizations (e.g., inlining)

```
1  module data_type_mod
2    implicit none
3
4    type data_type_t
5      ... ! Data components of data_type_t
6    contains
7      ... ! TBPs of data_type_t
8      procedure, non_overridable :: f
9    end type data_type_t
10
11 contains
12   ... ! Declare + implement subroutine that implements f
13 end module data_type_mod
```

```
1 subroutine poly_subroutine(dt)
2   use data_type_mod
3   implicit none
4   class(data_type_t), intent(in) :: dt
5   call dt%f()
6 end subroutine poly_subroutine
```

Outline

Polymorphism

- **Polymorphism** is a term used in sw development to describe a variety of techniques employed by programmers in order to create **flexible** and **reusable** sw components. Greek term, loosely translates to "many forms"
- In programming languages, a polymorphic entity is a variable or procedure that can store or operate on values of differing types **during the program's execution**
- Since polymorphic entities can operate on a wide range of values/types, they can also be used in a variety of programs, sometimes with **little or no change by the programmer**

Polymorphism in F2003

- The `class` keyword allows programmers to declare a variable as polymorphic
- A polymorphic variable is a variable whose data type is **dynamic at runtime**
- A polymorphic variable MUST BE either a **pointer**, an **allocatable**, or a **dummy argument**
- "is a" relationship helps visualizing how polymorphic variables interact with type extensions
- In the example, `person_p` can be a pointer to `person_t` or any of its type extensions, i.e., as long as the type of the pointer target "is a" `person_t`

```
1 type(person_t)    , target  :: person     ! Declaration of static type instances
2 type(employee_t)  , target  :: employee   ! (potential targets to person_p polymorhic pointer)
3 type(manager_t)   , target  :: manager
4
5 class(person_t)   , pointer :: person_p   ! Declaration of polymorphic pointer variable
6 person_p => person                        ! The data type of person_p is dynamic at runtime
7 person_p => employee                      ! (i.e., first person_t, then employee_t, and
8 person_p => manager                       ! finally, manager_t)
```

Polymorphism in F2003 (continued)

There are actually two basic forms of polymorphism:

- **Procedure polymorphism**: procedures that can operate on a variety of data types and values
- **Data polymorphism** : program variables that can store and operate on a variety of data types and values

Outline

Procedure polymorphism

- Procedure polymorphism arises when a procedure, such as function or subroutine, can take a variety of data types as arguments
- In F2003 accomplished when a procedure has one or more dummy arguments declared with the `class` keyword

```
1  ! Subroutine declaration
2  subroutine set_age(person, age)
3    implicit none
4    class(person_t), intent(inout) :: person
5    integer        , intent(in)    :: age
6    person%age = age
7  end subroutine set_age
8
9  ! Subroutine usage
10 type(person_t)     :: person
11 type(employee_t)   :: employee
12 type(manager_t)    :: manager
13
14 call set_age(person  , 35)
15 call set_age(employee, 26)
16 call set_age(manager , 50)
17
18 ... ! Set up the rest of components of
19     ! the three objects above
```

- `set_age` is polymorphic, `person` dummy argument is declared with `class` keyword
- The subroutine can operate on data types that satisfy the "is a" relationship (Lines 14-16)
- However, by default, it can only operate with components of the "declared" type of the polymorphic argument
- In the example, `person_t` is the declared type of `person`. Thus, by default, `set_age` can access `age`/`gender`
- What if the programmer needs to access to the components of the dynamic type of `person`? (e.g., if it is `employee_t` to `salary`?)
- 2003 offers the so-called `select type` construct, a.k.a. RTTI (run-time type identification)

Procedure polymorphism (continued)

```
1  subroutine initialize(person, age, gender, &
2                        salary, bonus)
3    implicit none
4    class(person_t)  , intent(inout)  :: person
5    integer          , intent(in)     :: age
6    integer          , intent(in)     :: gender
7    integer, optional, intent(in)     :: salary
8    integer, optional, intent(in)     :: bonus
9
10   person%age  = age
11   person%gender = gender
12
13   select type (person)
14   type is (person_t)
15      ! no further initialization required
16   class is (employee_t)
17      ! employee or manager required specific inits
18      if (present(salary))  then
19         person%salary = salary
20      else
21         person%salary = 0.0
22      endif
23   class default
24      stop 'unexpected dynamic type!'
25   end select
26
27  select type (person)
28   type is (manager_t)
29     if (present(bonus))  then
30         person%bonus = bonus
31     else
32         person%bonus = 0.0
33     endif
34  end select
35
36 end subroutine initialize
```

- Initialization procedure for objects in the data type hierarchy rooted at `person_t`
- It takes a polymorphic dummy argument of declared type `person_t`, and a set of initial values for the components of `person_t`
- Two optional dummy arguments are declared when a `employee_t` or `manager_t` is to be initialized
- I.e., when the dynamic type `person` is either `employee_t` or `manager_t`

Procedure polymorphism (continued)

```
1  subroutine initialize(person, age, gender, &
2                        salary, bonus)
3    implicit none
4    class(person_t)  , intent(inout)  :: person
5    integer          , intent(in)     :: age
6    integer          , intent(in)     :: gender
7    integer, optional, intent(in)     :: salary
8    integer, optional, intent(in)     :: bonus
9
10   person%age  = age
11   person%gender = gender
12
13   select type (person)
14   type is (person_t)
15      ! no further initialization required
16   class is (employee_t)
17      ! employee or manager required specific inits
18      if (present(salary))  then
19         person%salary = salary
20      else
21         person%salary = 0.0
22      endif
23   class default
24      stop 'unexpected dynamic type!'
25   end select
26
27  select type (person)
28   type is (manager_t)
29     if (present(bonus))  then
30         person%bonus = bonus
31     else
32         person%bonus = 0.0
33     endif
34  end select
35
36 end subroutine initialize
```

- The `select type` construct allows us to perform a type check on an object
- Two kind of type checks that we can perform: `type is` and `class is`
- `type is` is satisfied when the dynamic type of the polymorphic variable matches the one specified in parentheses
- `class is` is satisfied when the dynamic type of the polymorphic variable matches the one specified in parentheses, or if the former is a type extension of the latter

Procedure polymorphism (continued)

```
1  subroutine initialize(person, age, gender, &
2                        salary, bonus)
3    implicit none
4    class(person_t)  , intent(inout)  :: person
5    integer          , intent(in)     :: age
6    integer          , intent(in)     :: gender
7    integer, optional, intent(in)     :: salary
8    integer, optional, intent(in)     :: bonus
9
10   person%age  = age
11   person%gender = gender
12
13   select type (person)
14   type is (person_t)
15      ! no further initialization required
16   class is (employee_t)
17      ! employee or manager required specific inits
18      if (present(salary))  then
19         person%salary = salary
20      else
21         person%salary = 0.0
22      endif
23   class default
24      stop 'unexpected dynamic type!'
25   end select
26
27  select type (person)
28   type is (manager_t)
29     if (present(bonus))  then
30         person%bonus = bonus
31     else
32         person%bonus = 0.0
33     endif
34  end select
35
36 end subroutine initialize
```

- In the example, we will initialize `salary` when the dynamic type of `person` is either `employee_t` or `manager_t` (see Lines 16-22)
- If the dynamic type of `person` is NOT `person_t`, `employee_t`, or `manager_t`, we will execute the `class default` branch
- This branch may get executed if we added a new type extension of `person_t` without modifying the `initialize` subroutine (inherently not extensible subroutine!)
- In the example, why do you think that the `type is(person_t)` branch is needed? (even though it does not perform anything?)

Outline

Data polymorphism

- The `class` keyword lets F2003 programmers to declare polymorphic variables
- A polymorphic variable has a declared type and a dynamic type
- The dynamic type of a polymorphic variable is **determined at runtime**, and thus can change in the course of the program's execution
- A polymorphic variable must be either a **pointer**, an **allocatable**, or a **dummy argument**

```
1  subroutine initialize_person(person,...)
2    implicit none
3    ! Polymorphic dummy argument
4    class(person_t), intent(inout) :: person
5
6    ! Polymorphic pointer local variable
7    class(person_t), pointer :: person_p
8
9    ! Polymorphic allocatable local variable
10   class(person_t), allocatable :: person_a
11
12   ... ! Subroutine body
13 end subroutine initialize_person
```

- `person`, `person_p`, and `person_a` are polymorphic variables
- Each can hold values of type `person_t` or **any of its type extensions**
- The `person` dummy argument receives its dynamic type and value from the actual argument provided on a call to the `initialize_person` subroutine
- While polymorphic dummy arguments are the basis of procedure polymorphism, **pointer** and **allocatable** polymorphic variables are the one of **data polymorphism**

Polymorphic pointer variables

```
1  subroutine initialize_person(person,...)
2    implicit none
3    ! Polymorphic dummy argument
4    class(person_t), target, intent(inout) :: person
5
6    ! Polymorphic pointer local variable
7    class(person_t), pointer :: person_p
8
9    select type(person)
10     type is (person_t)
11       person_p => person
12       ... ! person specific code
13     type is (employee_t)
14       person_p => person
15       ... ! employee specific code
16     type is(manager_t)
17       person_p => person
18       ... ! manager specific code
19     class default
20       nullify(person_p)
21     end select
22 end subroutine initialize_person
```

- The polymorphic **pointer** variable `person_p` can point to an object of type `person_t` or **any of its type extensions**
- We used a `select type` statement on the dynamic type of `person` to emphasize that that the one of `person_p` can be of several types
- Either `person_t`, `employee_t` or `manager_t` in our particular example
- The dynamic type of `person_p` is not known until it is associated with a target (i.e., until `person_p => person` is executed)

Polymorphic allocatable variables

- The dynamic type of a polymorphic **allocatable** variable is determined at the point in which it is allocated through the `allocate` statement

```
1 class(person_t), allocatable :: person_a
2 allocate(person_a)
```

- This code example allocates a polymorphic variable `person_a`
- If we just provide the polymorphic variable name to `allocate`, then its dynamic type simply becomes its declared type
- Thus, in our example, the dynamic type of `person_a` becomes `person_t` after `allocate` statement
- *What if we want to specify a dynamic type different from the declared type?*

Polymorphic allocatable variables

- The dynamic type of a polymorphic **allocatable** variable is determined at the point in which it is allocated through the `allocate` statement

```
1 class(person_t), allocatable :: person_a
2 allocate(employee_t :: person_a) ! Typed allocation
```

- F2003 offers **typed allocation** to programmers to explicitly specify a dynamic type different from the declared type in an `allocate` statement
- A type name, followed by `::`, and the polymorphic allocatable variable name, is the syntax for typed allocation
- In the example, `employee_t` becomes the dynamic type of `person_a`
- Please note that `person_t` is still the declared type of `person_a`. As always, type specified in `allocate` MUST BE either declared type or any of its type extensions

Polymorphic allocatable variables

- 2 more involved examples of **typed allocation** (clone dynamic type and full instance)
- Nested `select type` statement required by `clone_full_instance` to ensure that the type of left and right hand side instances on intrinsic assignment (=) matches

```
1  subroutine clone_dynamic_type(person,person_a)
2   implicit none
3   ! Polymorphic dummy argument
4   class(person_t), intent(in)    :: person
5   ! Polymorphic allocatable dummy argument
6   class(person_t), allocatable, &
7                    intent(inout) :: person_a
8   select type(person)
9    type is (person_t)
10     allocate(person_t :: person_a)
11   type is (employee_t)
12     allocate(employee_t :: person_a)
13   type is(manager_t)
14     allocate(manager_t :: person_a)
15  end select
16 end subroutine clone_dynamic_type
```

Clone dynamic type

```
1  subroutine clone_full_instance(person,person_a)
2   implicit none
3   class(person_t), intent(in)    :: person
4   class(person_t), allocatable, &
5                    intent(inout) :: person_a
6   select type(person)
7    type is (person_t)
8      allocate(person_t :: person_a)
9      select type(person_a)
10     type is (person_t)
11        person_a = person
12     end select
13   type is (employee_t)
14     allocate(employee_t :: person_a)
15     select type(person_a)
16     type is (employee_t)
17        person_a = person
18     end select
19   type is(manager_t)
20     allocate(manager_t :: person_a)
21     select type(person_a)
22     type is (employee_t)
23        person_a = person
24     end select
25  end select
26 end subroutine clone_full_instance
```

Clone full instance

Polymorphic allocatable variables

- These examples, although interesting, **do not scale well** with # of type extensions
- Besides, they are inherently **non-extensible** as they MUST BE updated each time we add a new type extension of `person_t`
- Fortunately, F200X offers native support to overcome these issues

```
1  subroutine clone_dynamic_type(person,person_a)
2   implicit none
3   ! Polymorphic dummy argument
4   class(person_t), intent(in)    :: person
5   ! Polymorphic allocatable dummy argument
6   class(person_t), allocatable, &
7                    intent(inout) :: person_a
8   select type(person)
9    type is (person_t)
10     allocate(person_t :: person_a)
11   type is (employee_t)
12     allocate(employee_t :: person_a)
13   type is(manager_t)
14     allocate(manager_t :: person_a)
15  end select
16 end subroutine clone_dynamic_type
```

Clone dynamic type

```
1  subroutine clone_full_instance(person,person_a)
2   implicit none
3   class(person_t), intent(in)    :: person
4   class(person_t), allocatable, &
5                    intent(inout) :: person_a
6   select type(person)
7    type is (person_t)
8      allocate(person_t :: person_a)
9      select type(person_a)
10     type is (person_t)
11        person_a = person
12     end select
13   type is (employee_t)
14     allocate(employee_t :: person_a)
15     select type(person_a)
16     type is (employee_t)
17        person_a = person
18     end select
19   type is(manager_t)
20     allocate(manager_t :: person_a)
21     select type(person_a)
22     type is (employee_t)
23        person_a = person
24     end select
25  end select
26 end subroutine clone_full_instance
```

Clone full instance

Polymorphic allocatable variables

- F200X offers the **optional** `mold=` and `source=` dummy arguments of `allocate`
- Referred as mold and sourced allocation, respectively
- Semantics are equivalent to "Clone dynamic type" and "Clone full instance"
- The declared type of the actual argument to `mold=` and `source=` should match the one of the polymorphic variable being allocated, or be a type extension of it
- This actual argument does not have to be necessarily polymorphic (i.e., it can be `type` instead of `class`)

```
1  subroutine clone_dynamic_type(person,person_a)
2   implicit none
3   ! Polymorphic dummy argument
4   class(person_t), intent(in)    :: person
5   ! Polymorphic allocatable dummy argument
6   class(person_t), allocatable, &
7                    intent(inout) :: person_a
8
9   ! Select the dynamic type of person_a to
10  ! match the one of person
11  allocate(person_a, mold=person)
12 end subroutine clone_dynamic_type
```

Clone dynamic type

```
1  subroutine clone_full_instance(person,person_a)
2   implicit none
3   ! Polymorphic dummy argument
4   class(person_t), intent(in)    :: person
5   ! Polymorphic allocatable dummy argument
6   class(person_t), allocatable, &
7                    intent(inout) :: person_a
8
9   ! Select the dynamic type of person_a to
10  ! match the one of person, copy contents of
11  ! the latter into the former
12  allocate(person_a, source=person)
13 end subroutine clone_full_instance
```

Clone full instance

Unlimited polymorphism

- Polymorphism so far restricted to derived types and their type extensions
- This satisfies most applications, but sometimes it might be useful to have a procedure/variable that can operate on **any intrinsic or derived type** (scalar)
- To this end, F2003 offers **unlimited polymorphic** procedures and variables

```
1  subroutine initialize(unlim_poly_arg,...)
2    implicit none
3    ! Unlimited polymorphic dummy argument
4    class(*), intent(inout) :: unlim_poly_arg
5
6    ! Unlimited polymorphic pointer local variable
7    class(*), pointer :: unlim_poly_var_p
8
9    ! Unlimited polymorphic local variable
10   class(*), allocatable :: unlim_poly_var_a
11
12   ... ! Subroutine body
13 end subroutine initialize
```

- The `class(*)` keyword is used to declare an unlimited polymorphic variable
- An unlimited polymorphic variable must be either a **pointer**, an **allocatable**, or a **dummy argument** (just like "limited" polymorphic ones!)
- Indeed, working with unlimited polymorphic variables is very similar to working with "limited" ones; unlimited counterparts can operate on any intrinsic or derived type though

Unlimited polymorphism

- Two examples of unlimited polymorphic procedures to initialize dummy argument
- In the example on the right, an unlimited polymorphic pointer is assigned to an unlimited polymorphic target (see Line 10), and then a `select type` statement is used to query the dynamic type of the pointer
- We stress, nevertheless, that **any pointer or target** can be assigned to an unlimited polymorphic pointer

```
1  subroutine init_dummy_arg(poly_dummy_arg)
2   implicit none
3   ! Unlimited polymorphic dummy argument
4   class(*), intent(inout) :: poly_dummy_arg
5
6   select type (poly_dummy_arg)
7    type is (person_t)
8      ... ! person_t specific code here
9    type is (integer)
10     ... ! integer specific code here
11   type is(double precision)
12     ... ! double precision specific code here
13   ...
14  class default
15     stop 'init_dummy_arg: Unexpected type!'
16  end select
17 end subroutine init_dummy_arg
```

Direct dummy arg. initialization

```
1  subroutine init_dummy_arg(poly_dummy_arg)
2   implicit none
3   ! Unlimited polymorphic dummy argument
4   class(*), target, intent(inout) &
5             :: poly_dummy_arg
6
7   ! Unlimited polymorphic pointer
8   class(*), pointer :: poly_dummy_arg_p
9
10  poly_dummy_arg_p => poly_dummy_arg
11
12  select type (poly_dummy_arg_p)
13   type is (person_t)
14     ... ! person_t specific code here
15   type is (integer)
16     ... ! integer specific code here
17   type is(real)
18     ... ! real specific code here
19   ...
20  class default
21     stop 'init_dummy_arg: Unexpected type!'
22  end select
23 end subroutine init_dummy_arg
```

Dummy arg. initialization through pointer

Unlimited polymorphism

- Unlimited polymorphic variables can also be allocated with **typed allocation**
- Indeed, a type MUST BE specified with typed allocation, as there is no notion of declared type (i.e., default type) with `class(*)` variables
- Any F2003 type, intrinsic or derived, can be specified with typed allocation

```
1  subroutine clone_dynamic_type(person,person_a)
2   implicit none
3   ! Unlimited polymorphic dummy argument
4   class(*), intent(in)     :: person
5   ! Unlimited polymorphic allocatable dummy
          argument
6   class(*), allocatable, &
7             intent(inout) :: person_a
8   select type(person)
9    type is (person_t)
10     allocate(person_t :: person_a)
11   type is (integer)
12     allocate(integer :: person_a)
13   type is(real)
14     allocate(real :: person_a)
15  end select
16 end subroutine clone_dynamic_type
```

Clone dynamic type

```
1  subroutine clone_full_instance(person,person_a)
2   implicit none
3   ! Unlimited polymorphic dummy argument
4   class(*), intent(in)     :: person
5   ! Unlimited polymorphic allocatable dummy
          argument
6   class(*), allocatable, &
7             intent(inout) :: person_a
8   select type(person)
9    type is (person_t)
10     allocate(person_t :: person_a)
11     select type(person_a)
12     type is (person_t)
13        person_a = person
14     end select
15   type is (integer)
16     allocate(integer :: person_a)
17     select type(person_a)
18     type is (integer)
19        person_a = person
20     end select
21   type is (real)
22     ... ! Specific code for real
23  end select
24 end subroutine clone_full_instance
```

Clone full instance

Unlimited polymorphism

- We can also use `mold=` or `source=` with unlimited polymorphic variables!

```
1  subroutine clone_dynamic_type(person,person_a)
2   implicit none
3   ! Unlimited polymorphic dummy argument
4   class(*), intent(in)     :: person
5   ! Unlimited polymorphic allocatable dummy
          argument
6   class(*), allocatable, &
              intent(inout) :: person_a
8
9   ! Select the dynamic type of person_a to
10  ! match the one of person
11  allocate(person_a, mold=person)
12 end subroutine clone_dynamic_type
```

Clone dynamic type

```
1  subroutine clone_full_instance(person,person_a)
2   implicit none
3   ! Unlimited polymorphic dummy argument
4   class(*), intent(in)     :: person
5   ! Unlimited polymorphic allocatable dummy
          argument
6   class(*), allocatable, &
              intent(inout) :: person_a
8
9   ! Select the dynamic type of person_a to
10  ! match the one of person, copy contents of
11  ! the latter into the former
12  allocate(person_a, source=person)
13 end subroutine clone_full_instance
```

Clone full instance

- If the variable provided to `mold=` or `source=` is of type `class(*)`, the variable to be allocated MUST ALSO BE of type `class(*)`
- If the variable to be allocated is of type `class(*)`, then the one provided to `mold=` or `source=` can be of any type, i.e., `class(*)` and any derived or intrinsic type

```
1  class(*), allocatable :: my_poly_var_a
2  real                  :: my_real_var
3  my_real_var = 7.3
4  ! Sourced allocation with intrinsic type
5  allocate(my_poly_var, source=my_real_var)
```

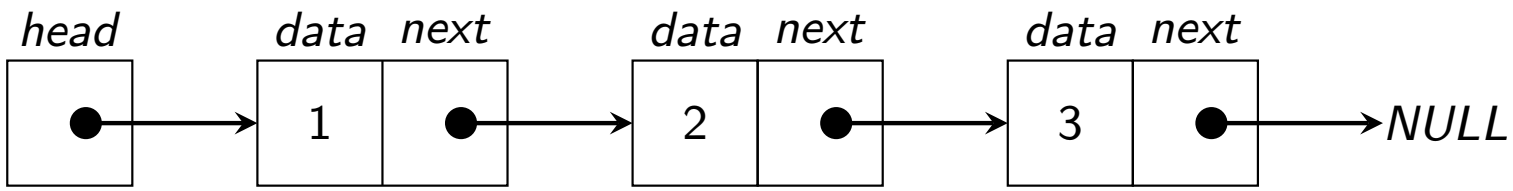
Remark

The examples in the following set of slides are available, together with GNU Make scripts for autocompilation, at a public Git repository that can be cloned as:

Execute me on a Linux terminal

```
$ git clone https://gitlab.com/femparadmin/fmw_oop_bcn_material.git
```

Merging alltogether: an unlimited polymorphic linked list



- We are going to develop a linked list data structure that can be used to hold data of **heterogeneous data types** to demonstrate the potential of unlimited polymorphism
- Traditionally, linked lists are such that all link list nodes have data of the same data type (e.g., a linked list of integers, a linked list of reals, etc.)
- Let us start by defining a derived type to represent each node in our linked list:

```
1 type node_list_t
2    class(*)          , pointer :: data  => null()
3    type(node_list_t), pointer :: next  => null()
4 end type node_list_t
```

  where `data` is an unlimited polymorphic data component that points to the data hold by the node, and `next` points to the next node in the linked list

- Recall that we do not want to expose internal details of this data structure to its users (information hiding). Let us thus place the data type in its own module, add a constructor, some TBPs to perform manipulations on it, and judiciously use `public`/ `private` (see next slide)

Merging alltogether: an unlimited polymorphic linked list

```
1  module node_list_mod
2    implicit none
3    private
4
5    type node_list_t
6      private
7      class(*)          , pointer :: data
8      type(node_list_t), pointer :: next
9    contains
10     procedure, non_overridable :: get_data
11     procedure, non_overridable :: get_next
12     procedure, non_overridable :: set_next
13     procedure, non_overridable :: free
14   end type node_list_t
15
16   interface node_list_t
17     module procedure construct_node_list
18   end interface node_list_t
19
20   public :: node_list_t
21 contains
22   ...
23 end module node_list_mod
```

```
1  ...
2  contains
3    function get_data(this)
4      class(node_list_t), intent(in) :: this
5      class(*), pointer :: get_data
6      get_data => this%data
7    end function get_data
8
9    function get_next(this)
10     class(node_list_t), intent(in) :: this
11     type(node_list_t), pointer :: get_next
12     get_next => this%next
13   end function get_next
14
15   subroutine set_next(this,next)
16     class(node_list_t), intent(inout) :: this
17     type(node_list_t), pointer :: next
18     this%next => next
19   end subroutine set_next
20   ...
```

- Due to the usage of `private` above, `node_list_t` users must use the `get_data` TBP to retrieve the data of a node in the list, `get_next` to get a pointer to next node, `set_next` to add a new node after a node, and `free` to deallocate all dynamic memory of the list headed at a node
- Note that the `get_data` function-like TBP returns a pointer to a unlimited polymorphic variable **such that it can return an object of arbitrary type**

Merging alltogether: an unlimited polymorphic linked list

```
1  module node_list_mod
2    implicit none
3    private
4
5    type node_list_t
6      private
7      class(*)          , pointer :: data
8      type(node_list_t), pointer :: next
9    contains
10     procedure, non_overridable :: get_data
11     procedure, non_overridable :: get_next
12     procedure, non_overridable :: set_next
13     procedure, non_overridable :: free
14   end type node_list_t
15
16   interface node_list_t
17     module procedure construct_node_list
18   end interface node_list_t
19
20   public :: node_list_t
21 contains
22   ...
23 end module node_list_mod
```

```
1  ...
2  contains
3   recursive subroutine free(this)
4     class(node_list_t), intent(inout) :: this
5     if ( associated(this%next) ) then
6       call this%next%free()
7       deallocate(this%next)
8     end if
9     deallocate(this%data)
10  end subroutine free
11
12  function construct_node_list(data)
13    class(*), intent(in) :: data
14    type(node_list_t), pointer :: &
15                        construct_node_list
16    allocate(construct_node_list)
17    allocate(construct_node_list%data, &
18             source=data)
19    nullify(construct_node_list%next)
20  end function construct_node_list
21 end module node_list_mod
```

- We employed type overloading to construct a new node (Recall that this lets us to hide a constructor function behind the name of the type itself)
- It returns a pointer to a newly allocated target (of type `node_list_t`), which the caller becomes responsible to deallocate later on
- Let us illustrate the usage of this data type with an example (next slide)

Merging alltogether: an unlimited polymorphic linked list

```
1  program node_list_program
2    use node_list_mod
3    implicit none
4    type(node_list_t), pointer :: node
5    class(*), pointer :: data
6    type(node_list_t), pointer :: current
7    integer :: i
8
9    ! Create an integer linked list with
10   ! 10 nodes, and data 1, 2, ..., 10
11   ! node becomes the head of the list
12   node    => node_list_t(1)
13   current => node
14   do i=2, 10
15      call current%set_next(node_list_t(i))
16      current => current%get_next()
17   end do
18
19   ! Print contents of link list nodes
20   ! to standard output
21   current => node
22   do while ( associated(current) )
23      data => current%get_data()
24      select type (data)
25      type is (integer)
26         write(*,*) data
27      end select
28      current => current%get_next()
29   end do
30
31   ! Free all dynamic memory
32   call node%free()
33   deallocate(node)
34 end program node_list_program
```

- Although functional, this example reveals that the user of type `node_list_t` has still to deal with many details regarding the construction and traversal of a linked list
- The real power of OOP lies in its ability to create **flexible** and **reusable** components
- With `node_list_t` in isolation, a lot of (very prone to errors) code will be replicated
- Therefore, we can create another object, say `list_t`, that acts as the front-end with the user and hides all the details underlying `node_list_t` (see next slide)

Merging alltogether: an unlimited polymorphic linked list

```
1  module list_mod
2    use node_list_mod
3    implicit none
4
5    type list_t
6      private
7      type(node_list_t), pointer :: head => NULL()
8      type(node_list_t), pointer :: tail => NULL()
9    contains
10     procedure, private :: push_back_integer
11     procedure, private :: push_back_real
12     procedure, private :: push_back_logical
13     procedure, private :: push_back_data
14
15     generic            :: push_back =>       &
16                           push_back_integer, &
17                           push_back_real,    &
18                           push_back_logical
19
20     procedure          :: print
21     procedure          :: free
22   end type list_t
23
24 contains
25   ... ! module procedures for TBPs above
26 end module list_mod
```

- `list_t` has 2 (`private`) data components, `head`/`tail`, pointing to first/last node of the linked list, resp.
- `tail` let us to easily add new data to the end of the list
- Next, we have three (`private`) TBPs, `push_back_integer`, `..._real`, and `..._logical`
- The `push_back_data` TBP let us to push back `class(*)` data to the linked list, and acts as the main subroutine on which the other three rely on

Merging alltogether: an unlimited polymorphic linked list

```
1  subroutine push_back_integer(this, data)
2    implicit none
3    class(list_t), intent(inout) :: this
4    integer      , intent(in)    :: data
5    call this%push_back_data(data)
6  end subroutine push_back_integer
7
8  subroutine push_back_data(this,data)
9    implicit none
10   class(list_t), intent(inout) :: this
11   class(*)     , intent(in)    :: data
12   if ( .not. associated(this%head)) then
13     this%head => node_list_t(data)
14     this%tail => this%head
15   else
16     call this%tail%set_next(node_list_t(data))
17     this%tail => this%tail%get_next()
18   end if
19 end subroutine push_back_data
```

- The `push_back_integer` subroutine takes a `list_t` and an integer value and just calls the `push_back_data` TBP
- The only diff among `push_back_integer` and `..._real`, and `..._logical` is the data type of the dummy argument `data`

Merging alltogether: an unlimited polymorphic linked list

```
1  subroutine push_back_integer(this, data)
2    implicit none
3    class(list_t), intent(inout) :: this
4    integer      , intent(in)    :: data
5    call this%push_back_data(data)
6  end subroutine push_back_integer
7
8  subroutine push_back_data(this,data)
9    implicit none
10   class(list_t), intent(inout) :: this
11   class(*)     , intent(in)    :: data
12   if ( .not. associated(this%head)) then
13     this%head => node_list_t(data)
14     this%tail => this%head
15   else
16     call this%tail%set_next(node_list_t(data))
17     this%tail => this%tail%get_next()
18   end if
19 end subroutine push_back_data
```

- The `push_back_data` subroutine takes a `list_t` and a `class(*)` value
- If `head` is not associated, we add `data` to the start of the list by assigning `head` to a newly created linked list node
- Otherwise, we add it after `tail`, while properly updating it afterwards

Merging alltogether: an unlimited polymorphic linked list

```
1  module list_mod
2    use node_list_mod
3    implicit none
4
5    type list_t
6      private
7      type(node_list_t), pointer :: head => NULL()
8      type(node_list_t), pointer :: tail => NULL()
9    contains
10     procedure, private :: push_back_integer
11     procedure, private :: push_back_real
12     procedure, private :: push_back_logical
13     procedure, private :: push_back_data
14
15     generic            :: push_back =>       &
16                           push_back_integer, &
17                           push_back_real,    &
18                           push_back_logical
19
20     procedure          :: print
21     procedure          :: free
22   end type list_t
23
24 contains
25   ... ! module procedures for TBPs above
26 end module list_mod
```

- Going back to the `list_t` definition, we note the usage of the `generic` keyword, which lets us define a generic TBP
- These act pretty much likely generic interfaces, except that they are specified in the derived type and they can only be overloaded with TBPs
- We call `push_back` and either `push_back_integer` and `..._real`, and `..._logical` will be called
- The compiler determines the procedure to be invoked depending on the type of the actual argument passed to the generic TBP (also referred as **compilation-time polymorphism**)

Merging alltogether: an unlimited polymorphic linked list

And the final program is ...

```
1  program list_program
2    use list_mod
3    implicit none
4
5    type(list_t) :: list
6
7    ! Create an heterogeneous data type linked list
8    call list%push_back(3)
9    call list%push_back(4.45)
10   call list%push_back(.true.)
11
12   ! Print contents of link list nodes to standard output
13   call list%print()
14
15   ! Free all dynamic memory
16   call list%free()
17 end program list_program
```

Outline

Abstract Types and Deferred Bindings

- Fortran 2003 allows to define **abstract** data types
- An **abstract** data type is a **partially implemented data type** which subclasses have to complete via type extension
- An **abstract** data type provides reusable data and code to subclasses, and acts as a template to which its subclasses have to adhere to
- An **abstract** data type **CANNOT (thus) be instantiated**
- A type extension can also be declared **abstract**, but ultimately it must be extended by a non-abstract type if it is ever to be instantiated in a program

Abstract Types and Deferred Bindings

```
1  module abstract_type_mod
2   implicit none
3   private
4
5   type, abstract :: abstract_type_t
6    private
7    ... ! Data components
8   contains
9    procedure[,non_overridable] :: regular_TBP1
10   ... ! More regular TBPs (if applies)
11
12   procedure(def1_itfc), deferred :: def_TBP1
13   ... ! More deferred TBPs (if applies)
14  end type abstract_type_t
15
16  abstract interface
17   subroutine def1_itfc(this,arg1,arg2,...)
18    import :: abstract_type_t, ...
19    class(abstract_type_t), intent(in) :: this
20    ... ! Declaration of rest of args.
21   end subroutine def1_itfc
22   ...
23  end interface
24 contains
25  ...
26 end module abstract_type_mod
```

- An **abstract** data type includes the `abstract` keyword in its type definition
- Besides, it is composed of either regular or **deferred** TBPs
- **deferred** TBPs are NOT implemented in the abstract type, but **MUST BE implemented** in any of its non-abstract type extensions
- Deferred TBPs require an abstract interface associated with them

General syntax of deferred TBPs declaration

```
PROCEDURE [(interface-name)], deferred :: procedure-name
```

Unlimited polymorphic link list revisited

```
1  module list_mod
2    use node_list_mod
3    implicit none
4    private
5
6    type, abstract :: list_t
7     private
8     type(node_list_t), pointer :: head   => NULL()
9     type(node_list_t), pointer :: tail   => NULL()
10    type(node_list_t), pointer :: cursor => NULL()
11
12   contains
13    ! list_t construct and destruct TBPs
14    procedure, non_... :: push_back_data
15    procedure, non_overridable :: free
16
17    ! list_t traversal TBPs
18    procedure, non_overridable :: first
19    procedure, non_overridable :: next
20    procedure, non_overridable :: has_finished
21    procedure, non_... :: get_current_data
22    ...
23   end type list_t
24
25   public :: list_t
26 contains
27   ...
28
29 end module list_mod
```

- Let us factor most of the code of the unlimited polymorphic linked list on an `abstract` data type `list_t`
- Apart from `head` and `tail`, we added a data component `cursor` and a set of accompanying TBPs to let subclasses to sequentially traverse the list while preserving data hiding (**iterator** OO design pattern variant)
- In particular, `first()` positions `cursor` on the first node of the list, `next()` moves `cursor` to next node, and `has_finished()` tells whether `cursor` is already at the end of the list
- `get_current_data()` returns an unlimited polymorphic pointer to the data within node `cursor` is currently positioned on

Unlimited polymorphic link list revisited

```
1  subroutine first(this)
2    implicit none
3    class(list_t), intent(inout) :: this
4    this%cursor => this%head
5  end subroutine first
6
7  subroutine next(this)
8    implicit none
9    class(list_t), intent(inout) :: this
10   this%cursor => this%cursor%get_next()
11 end subroutine next
12
13 function has_finished(this)
14   implicit none
15   class(list_t), intent(in) :: this
16   logical :: has_finished
17   has_finished=.not. associated(this%cursor)
18 end function has_finished
19
20 function get_current_data(this)
21   implicit none
22   class(list_t), intent(in) :: this
23   class(*), pointer :: get_current_data
24   get_current_data => this%cursor%get_data()
25 end function get_current_data
```

- Let us factor most of the code of the unlimited polymorphic linked list on an `abstract` data type `list_t`
- Apart from `head` and `tail`, we added a data component `cursor` and a set of accompanying TBPs to let subclasses to sequentially traverse the list while preserving data hiding (**iterator** OO design pattern variant)
- In particular, `first()` positions `cursor` on the first node of the list, `next()` moves `cursor` to next node, and `has_finished()` tells whether `cursor` is already at the end of the list
- `get_current_data()` returns an unlimited polymorphic pointer to the data within node `cursor` is currently positioned on

Unlimited polymorphic link list revisited

```
1  module list_mod
2    use node_list_mod
3    implicit none
4    private
5    type, abstract :: list_t
6     private
7     ... ! Data components
8    contains
9     ... ! Regular TBPs
10
11    ! Deferred TBP to print list_t to stdout
12    procedure(print_list), deferred :: print
13   end type list_t
14
15   abstract interface
16    subroutine print_list(this)
17     import :: list_t
18     class(list_t), intent(inout) :: this
19    end subroutine print_list
20   end interface
21
22   public :: list_t
23 contains
24   ...
25 end module list_mod
```

- Going back to the type definition, the `print` TBP was declared **deferred**
- Any type extension is forced to develop a TBP that prints to standard output the contents of the list
- This makes sense provided that subclasses of `list_t` are actually the ones responsible to decide which data types(s) they are going to support
- In order to implement such TBP, subclasses have at their disposal the traversal mechanisms offered by `list_t` (see next slide)

Unlimited polymorphic link list revisited

```
1  ...
2   type, extends(list_t) :: int_list_t
3    private
4   contains
5    ! Child's TBPs to push back a new integer
6    ! and get current's node list integer
7    procedure :: push_back
8    procedure :: get_current
9
10   ! TBP overriding parent's deferred method
11   procedure :: print
12  end type int_list_t
13 ...
14  subroutine push_back(this,data)
15   implicit none
16   class(int_list_t), intent(inout) :: this
17   integer          , intent(in)    :: data
18   ! Call parent's unlimited polymorphic
19   ! variant of push_back
20   call this%push_back_data(data)
21  end subroutine push_back
22
23  function get_current(this)
24   implicit none
25   class(int_list_t), intent(in) :: this
26   integer :: get_current
27   class(*), pointer :: data
28   ! Call parent's unlimited polymorphic
29   ! variant of get_current
30   data => this%get_current_data()
31   select type(data)
32   type is(integer)
33     get_current = data
34   end select
35  end function get_current
36   ...
```

- We now define `int_list_t` as a type extension of `list_t`, a linked list of integers
- It defines specialized `push_back` and `get_data` TBPs to let the client work solely with integer data (i.e., s.t. he/she is not aware at all of unlimited polymorphism)

Unlimited polymorphic link list revisited

```
1  type, extends(list_t) :: int_list_t
2   private
3  contains
4   ...
5   ! TBP overriding parent's deferred method
6   procedure :: print
7  end type int_list_t
8
9  public :: int_list_t
10
11 contains
12  ...
13  subroutine print(this)
14   implicit none
15   class(int_list_t), intent(inout) :: this
16
17   call this%first()
18   do while (.not. this%has_finished() )
19      write(*,*) this%get_current()
20      call this%next()
21   end do
22  end subroutine print
```

- It (is forced to) implement(s) the `print` TBP that was declared **deferred** in its abstract parent
- The implementation of `print` in `list_t` uses the TBPs `first()`, `next()` and `has_finished()` (provided by its parent class) in order to control the traversal over the linked list nodes

Unlimited polymorphic link list revisited

And the final program is ...

```
1  program list_program
2    use int_list_mod
3    implicit none
4
5    type(int_list_t) :: list
6    ! type(list_t) :: list ! ILLEGAL !!!! WHY?
7
8    ! Fill integer data type linked list
9    call list%push_back(1)
10   call list%push_back(2)
11   call list%push_back(3)
12
13   ! Print contents of link list nodes
14   ! to standard output used overrided
15   ! version in int_list_t
16   call list%print()
17
18   ! Free all dynamic memory
19   call list%free()
20 end program list_program
```

Outline

Fortran 200X features not covered

- Procedure pointers
- Procedure pointers as data components of derived types
- Final TBPs
- Parameterized data types (partial support for generic programming)
- Operator overloading (combined with TBPs)

```
1 use operator_mod
2 type(operator_t) :: A,B,C
3 C = A*B + B
```

```
1  module operator_mod
2    private
3    type, abstract :: operator_t
4     contains
5       procedure(op2_interface) :: sum  => sum_operator
6       procedure(op2_interface) :: mult => mult_operator
7       generic :: operator(+) => sum
8       generic :: operator(*) => mult
9    end type operator_t
10
11   abstract interface
12      function op2_interface(x,y) result(z)
13        import :: operator_t
14        implicit none
15        class(operator_t), intent(in) :: x,y
16        class(operator_t) :: z
17      end function op2_interface
18   end interface
19
20 end module operator_mod
```

OO design patterns

- They represent the best practices used by experienced object-oriented software developers
- Reusable solutions to general problems that software developers faced during software development
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time
- The concept was introduced in the book titled "Design Patterns - Elements of Reusable Object-Oriented Software" by E. Gamma, R. Helm, R. Johnson and J. Vlissides, a.k.a. the Gang's of Four book (GoF)
- They can be classified into:
  - Creational patterns. Used to construct objects such that they can be decoupled from their implementing system. Single factory method (see exercise), ...
  - Structural patterns. Used to form large object structures between many disparate objects. Composite, ...
  - Behavioral patterns. Used to manage algorithms, relationships, and responsibilities between objects. Iterator (see linked list example),...
- Click here to download OO patterns quick reference guide

F2003 and F2008 compiler support

- **caveat**: before using a feature of the F2003/F2008 standard thoroughly in your sw project DO CHECK that the feature is supported by most common compilers available on high-end computing environments (otherwise your code won't be actually portable). In particular, go to:
  - F2003 compiler support
  - F2008 compiler support
- If the feature has been only very recently incorporated, it might have partial and/or fragile support. Check for compiler BUGs on compiler's mailing list archives if you observe a strong behaviour of your program that does not work as expected accordingly to the language standard

Outline

Fortran200X OOP projects freely available on the Internet

In our experience, exploring OOP designs and software from experienced scientific software engineers was tremendously useful to simplify the steep learning curve of OOP in Fortran200X. Fortunately, there are a number of high-quality Fortran200X open source software projects freely available on the Internet (here only a partial list):

- Fortran F/OSS programmers group. Not actually a software project, but a space to collaborate on new open source Fortran and Fortran related projects. Includes a place to collaborate on proposals for the next Fortran standard, available here
- Fortran Parameter List. An extensible parameter dictionary of `<key,value>` pairs, with `key` being a character string, and `value` any intrinsic or derived data type scalar or arbitrary rank array
- XH5For. XDMF parallel partitioned mesh I/O on top of HDF5
- FLAP. Fortran command Line Arguments Parser for poor people
- VTKFortran. Pure Fortran VTK (XML) API
- FortranParser. Fortran 2008 parser of mathematical expressions, based on Roland Schmehl `fparser`
- PFLOTRAN. A Massively Parallel Reactive Flow and Transport Model for describing Surface and Subsurface Processes
- FEMPAR. A Fortran200X OOP embarrassingly parallel multilevel finite element framework

References

- Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. Prentice Hall, 2008.
- The Clean Coder: A Code of Conduct for Professional Programmers. Robert C. Martin. Prentice Hall, 2011.
- Clean Architecture: A Craftsman's Guide to Software Structure and Design. Robert C. Martin. Prentice Hall, 2017.
- Design patterns: elements of reusable object-oriented software. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Addison Wesley professional computing series, 1995.
- Scientific Software Design: The Object-Oriented Way. Damian Rouson and Jim Xia. Cambridge University Press, 2011.
- The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures. Jeanne C. Adams, Walter S. Brainerd, Richard A. Hendrickson, Richard E. Maine, Jeanne T. Martin, Brian T. Smith. Springer, 2009
- Mark Leair Object Oriented Programming in Fortran2003 articles available here
- Fortran Wiki