Advanced Fortran Topics

Reinhold Bader

Gilbert Brietzke

Nisarg Patel

Leibniz Supercomputing Centre Munich

Fortran features under consideration

Continuing Standardization process:

	Fortran 66	ancient	
	Fortran 77 (1980)	traditional	
F95	Fortran 90 (1991)	large revision	
	Fortran 95 (1997)	small revision	
F03	**Fortran 2003** (2004)	large revision	
F08	**Fortran 2008** (2010)	mid-size revision	
	TS 29113 (2012)	extends C interop	integrated in F18
	TS 18508 (2015)	extends parallelism	
F18	Fortran 2018 (2018)	current standard	

Focus of this course is on F03 and F08

- some F18 features will be also covered

Overview of covered features

- **Day 1:**
  - the environment problem; object-based and object-oriented programming.
- **Day 2:**
  - further object-oriented features, advanced I/O topics, parameterized derived types
- **Day 3:**
  - interoperation with C, basics of Coarray programming
- **Day 4:**
  - Advanced coarray programming
- **Exercises:** interspersed with talks – see printed schedule
- **Prerequisites:**
  - **good** knowledge of F95
  - as covered e.g., in the winter event „Programming with Fortran“ (and some own experience, if possible)
  - some knowledge of OpenMP (shared memory parallelism)

Social Event and Guided Tour

- **Note:**
  - due to the remote participation, no social event is planned for this edition of the course

- ~~**If desired by participants:**~~
  - ~~joint dinner (self-funded) in the centre of Garching (Neuwirt) on **Monday evening** at 19:00~~

- ~~**Guided Tour through the computer rooms at LRZ**~~
  - ~~on **Wednesday** starting 18:00, approximately 60 minutes~~

Conventions and Flags used in these talks

- **Standards conformance**

✓ Recommended practice

? Standard conforming, but considered questionable style

⚠ Dangerous practice, likely to introduce bugs and/or non-conforming behaviour

💣 Gotcha! Non-conforming and/or definitely buggy

- **Implementation dependencies**

🔨 Processor dependent behaviour (may be unportable)

- **Performance**

🚀 language feature for performance

Some references

- **Modern Fortran explained (8th edition)**
  - Michael Metcalf, John Reid, Malcolm Cohen, OUP, 2018
- **The Fortran 2003 Handbook**
  - J. Adams, W. Brainerd, R. Hendrickson, R. Maine, J. Martin, B. Smith. Springer, 2008
- **Guide to Fortran 2008 programming (introductory text)**
  - W. Brainerd. Springer, 2015
- **Modern Fortran – Style and Usage (best practices guide)**
  - N. Clerman, W. Spector. Cambridge University Press, 2012
- **Scientific Software Design – The Object-Oriented Way (1st edition)**
  - Damian Rouson, Jim Xia, Xiaofeng Xu, Cambridge, 2011

References cont'd

- **Design Patterns – Elements of Reusable Object-oriented Software**
  - E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison-Wesley, 1994
- **Modern Fortran in Practice**
  - Arjen Markus, Cambridge University Press, 2012
- **Introduction to High Performance Computing for Scientists and Engineers**
  - G. Hager and G. Wellein
- **Download of (updated) PDFs of the slides and exercise archive**
  - freely available under a creative commons license
  - https://doku.lrz.de/display/PUBLIC/PRACE+Course%3A+Advanced+Fortran+Topics

The environment problem
and
some features from Fortran 2003

Recap: Module program unit

- **A program unit that permits packaging of**
  - procedure **interfaces**
  - **global** variables
  - named constants
  - **type definitions** (recall derived types)
  - **named** interfaces
  - procedure implementations

  for reuse, as well as supporting
  - **information hiding**
  - (limited) **namespace** management
- **Other program units access a module´s public entities**
  - **use association**

also known as **encapsulation**

Some F03 extensions for handling of globals

- **Example: Define entities which exist only once**
  - e.g. large arrays

```
MODULE mod_ptype
  IMPLICIT none
  PRIVATE
  INTEGER, PARAMETER :: pdim = 100
  TYPE, PRIVATE :: ptype

    REAL, PUBLIC :: field (pdim, pdim)
  END TYPE

  TYPE (ptype), PUBLIC :: o_ptype
END MODULE
```

type definition itself is only visible inside module

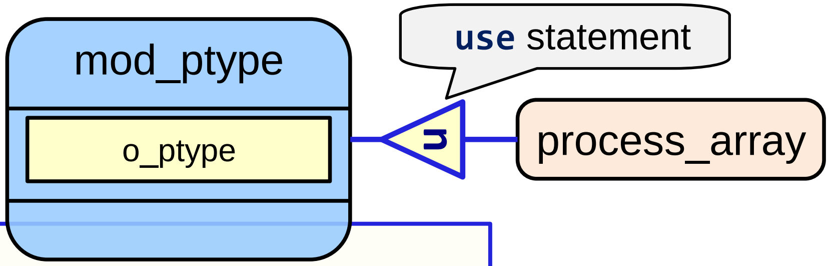
type components are accessible wherever an object is

object (a **global variable**) is accessible from program units that use the module

  - „**Singleton**“ programming pattern

Using the Singleton

- **Main program as example client:**



```
PROGRAM process_array
   USE mod_ptype
   ...
   TYPE(ptype) :: o2

   o_ptype % field(i, j) = 4
   ...
   x(:,:) = o_ptype % field(low:high,low:high)
```

type is private
→ will **not** compile

access to **o_ptype**
by use association

- client cannot create an object of the private type, but can access the (only) created object of that type

See **examples/singleton**

Global entities: Threading issues

- **Typical threading model used in HPC applications**
  - OpenMP, a directive based method for shared memory parallelism
- **What happens if global variables need to be accessed from threaded parts of the code?**
  How can „thread-safeness“ be achieved?
  1. Shared variables
     → use mutual exclusion to avoid data races, or
     → process arrays with work-sharing regions
  2. Threadprivate variables if needed for semantic reasons → may be problematic to use, especially across multiple parallel regions
- **Recommendation:**
  - avoid indiscriminate use of globals

The environment problem: setting the stage

- **Calculation of**

$$I = \int_a^b f(x,p)dx$$

where

- *f(x,p)* is a real-valued function of a real variable x and a variable p of some undetermined type
- a, b are real values

- **Tasks to be done:**
  - procedure with algorithm for establishing the integral → depends on the properties of *f(x,p)* (does it have singularities? etc.)

$$I \approx \sum_{i=1}^{n} w_i f(x_i, p)$$

  - function that evaluates *f(x,p)*

- **Case study provides a simple example of very common programming tasks with similar structure in scientific computing.**

Using a canned routine: D01AHF
(Patterson algorithm in NAG library)

- **Interface:**

requested precision

```
DOUBLE PRECISION FUNCTION d01ahf (a, b, epsr, npts, relerr, f, nlimit, ifail)
    INTEGER :: npts, nlimit, ifail
    DOUBLE PRECISION :: a, b, epsr, relerr, f
    EXTERNAL :: F
```

uses a function argument

```
DOUBLE PRECISION FUNCTION f (x)
    DOUBLE PRECISION :: x
```

(user-provided function)

- **Invocation:**

define a, b

```
:
res = d01ahf(a, b, 1.0e-11, &
   npts, relerr, my_fun, -1, is)
```

- **Mass-production of integrals**
  - may want to parallelize

```
!$omp parallel do
DO i=istart, iend
   : ! prepare
   res(i) = d01ahf(…, my_fun, …)
END DO
!$omp end parallel do
```

  - **need to check** library documentation: thread-safeness of `d01ahf`

Mismatch of user procedure implementation

- **User function may look like this:**

```
SUBROUTINE user_proc(x, n, a, result)
  REAL(dk), INTENT(in) :: x, a
  INTEGER, INTENT(in) :: n
  REAL(dk), INTENT(out) :: result
END SUBROUTINE
```

dk has the value **kind(1.0D0)**

do you remember what „**INTENT**“ means?

  - parameter „p“ is actually the tuple (n, a) → no language mechanism available for this

- **or like this**

```
REAL(dk) FUNCTION user_fun(x, p)
  REAL(dk), intent(in) :: x
  TYPE(p_type), intent(in) :: p
END FUNCTION
```

Compiler would accept this one due to the implicit interface for it, but it is likely to bomb at run-time

- **Neither can be used as an actual argument in an invocation of** `d01ahf`

Solution 1: Wrapper with global variables

```
MODULE mod_user_fun
  REAL(dk) :: par
  INTEGER :: n
contains
  FUNCTION arg_fun(x) result(r)
    REAL(dk) :: r, x
    CALL user_proc(x, n, par, r)
  END FUNCTION arg_fun
  :
END MODULE mod_user_fun
```

global variables (implies SAVE attribute)

has suitable interface for use with `d01ahf`

further procedures, e.g. `user_proc` itself

- **Usage:**

```
USE mod_user_fun

par = … ; n = …
res = d01ahf(…, arg_fun, …)
```

supply values for global variables

Disadvantages of Solution 1

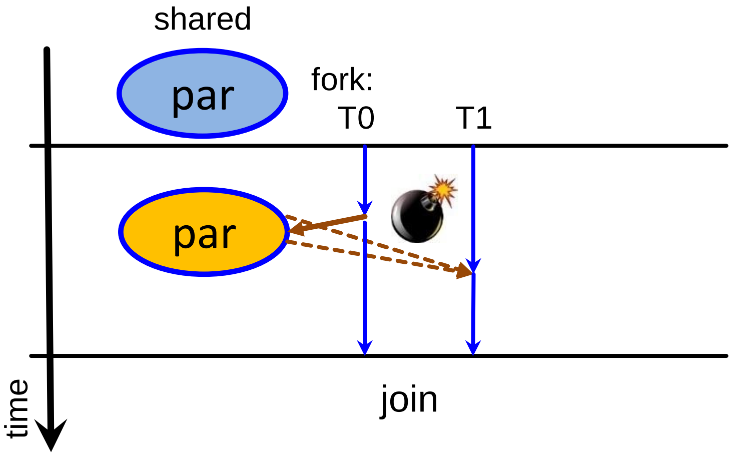
Additional function call overhead

- is usually not a big issue (nowaday‘s implementations are quite efficient, especially if no stack-resident variables must be created).

Solution not thread-safe (even if `d01ahf` itself is)

- expect differing values for **`par`** and **`n`** in concurrent calls:

```fortran
!$omp parallel do
DO i=istart, iend
   par = …; n = …
   res(i) = d01ahf(…, arg_fun, …)
END DO
!$omp end parallel do
```



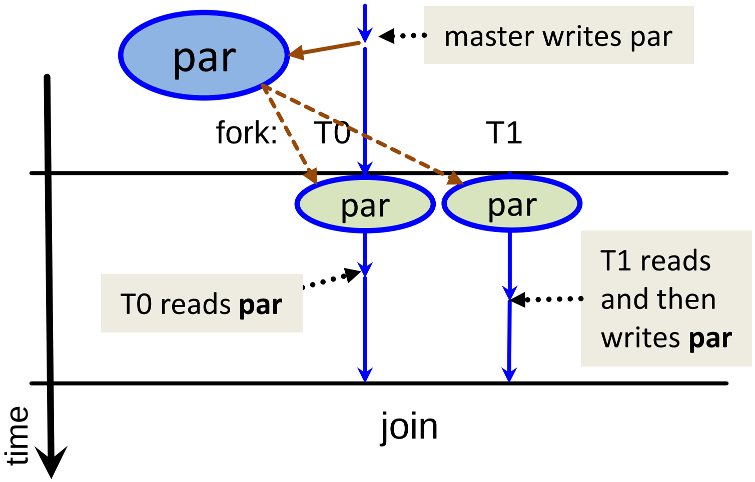
- results in unsynchronized access to the **shared** variables **`par`** and **`n`** from **different** threads → race condition → does not conform to the OpenMP standard → **wrong results**

Making Solution 1 thread-safe

Threadprivate storage

```
MODULE mod_user_fun
  REAL(dk) :: par
  INTEGER :: n
!$omp threadprivate (par, n)
   …
```

thread-individual copies
are created in parallel regions



Usage may require additional care as well

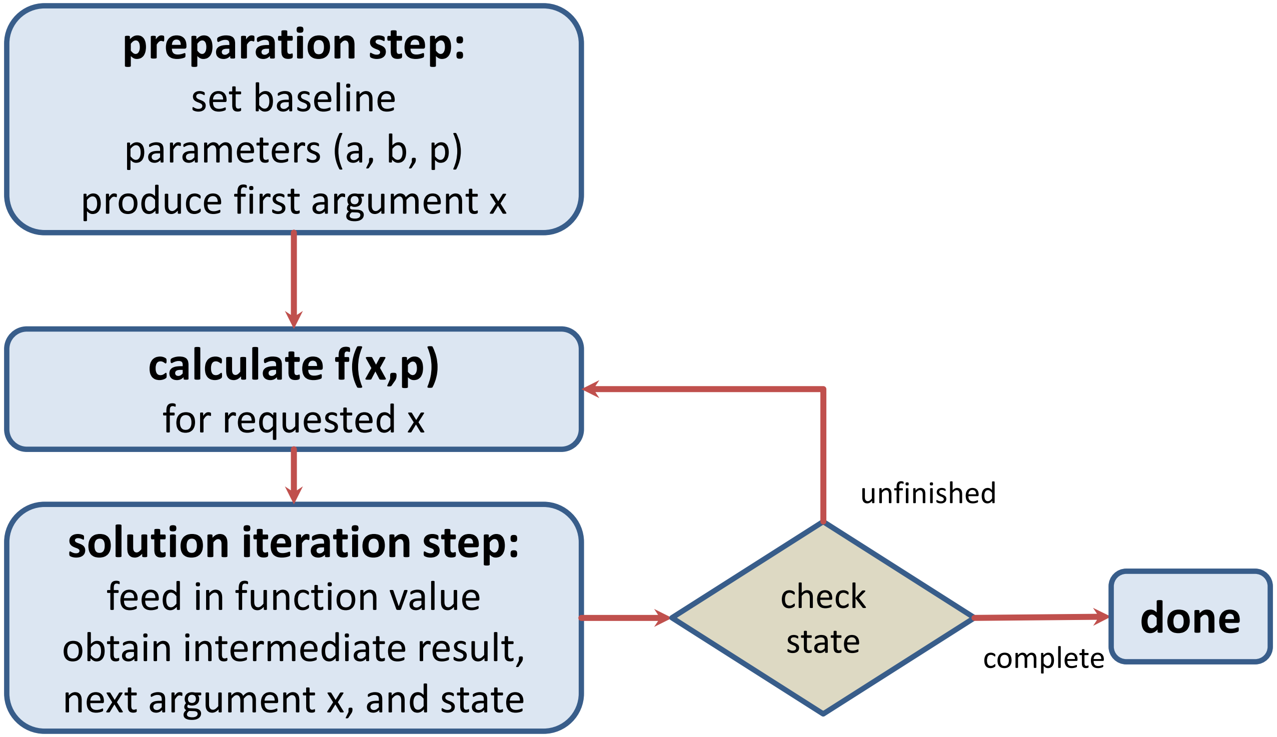
```
  par = …
!$omp parallel do copyin(par)
  DO i = istart, iend
    n = …
    … = d01ahf(…, arg_fun, …)
    IF (…) par = …
  END DO
!$omp end parallel do
```

broadcast from master copy
needed for par

A bit cumbersome:
non-local programming
style required

Solution 2: Reverse communication

- **Change design of integration interface:**
  - instead of a function interface, provider requests a function value
  - provider provides an argument for evaluation, and an exit condition



Solution 2: Typical example interface

Uses two routines:

```fortran
SUBROUTINE initialize_integration(a, b, eps, x)
  REAL(dk), INTENT(in)  :: a, b, eps
  REAL(dk), INTENT(out) :: x
END SUBROUTINE
SUBROUTINE integrate(fval, x, result, stat)
  REAL(dk), INTENT(in)    :: fval
  REAL(dk), INTENT(out)   :: x
  REAL(dk), INTENT(inout) :: result
  INTEGER,  INTENT(out)   :: stat
END SUBROUTINE
```

shall not be modified by caller while calculation iterates

- first is called once to initialize an integration process
- second will be called repeatedly, asking the client to perform further function evaluations
- final result may be taken once `stat` has the value `stat_continue`

Solution 2: Using the reverse communication interface

```
PROGRAM integrate
  :
  REAL(dk), PARAMETER :: a = 0.0_dk, b = 1.0_dk, eps = 1.0e-6_dk
  REAL(dk) :: x, result, fval, par
  INTEGER :: n, stat
  n = …; par = …
  CALL initialize_integration(a, b, eps, x)
  DO
    CALL user_proc(x, n, par, fval)
    CALL integrate(fval, x, result, stat)
    IF (stat /= stat_continue) EXIT
  END DO
  WRITE(*, '(''Result: '',E13.5,'' Status: '',I0)') result, stat
CONTAINS
  SUBROUTINE user_proc( … )
    :
  END SUBROUTINE user_proc
END PROGRAM
```

- avoids the need for interface adaptation and global variables
- some possible issues will be discussed in an exercise

Dynamic memory and features for object-based programming

Recapitulation: dynamic objects

Add a suitable attribute to an entity:

initial state is „unallocated“

```
REAL, ALLOCATABLE :: x(:)
```

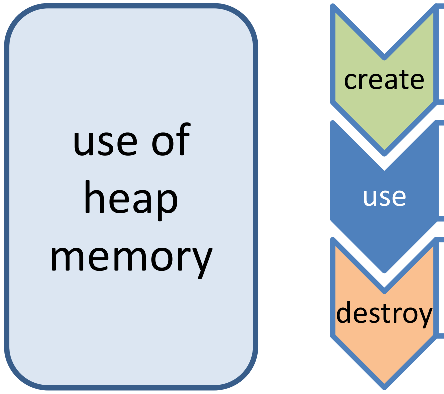
deferred shape

initial state is "unassociated"

```
REAL, POINTER :: p(:) => null()
```

Typical life cycle management:

non-default lower bounds are possible (use LBOUND and UBOUND intrinsics)



```
ALLOCATE(x(2:n), p(3), stat=my_status)
```

```
x(:) = ...
p(:) = ...
```

definitions and references

```
DEALLOCATE(x, p)
```

Status checking:

(hints at semantic differences!)

```
IF (allocated(x)) THEN; ...
```

```
IF (associated(p)) THEN; ...
```

logical functions

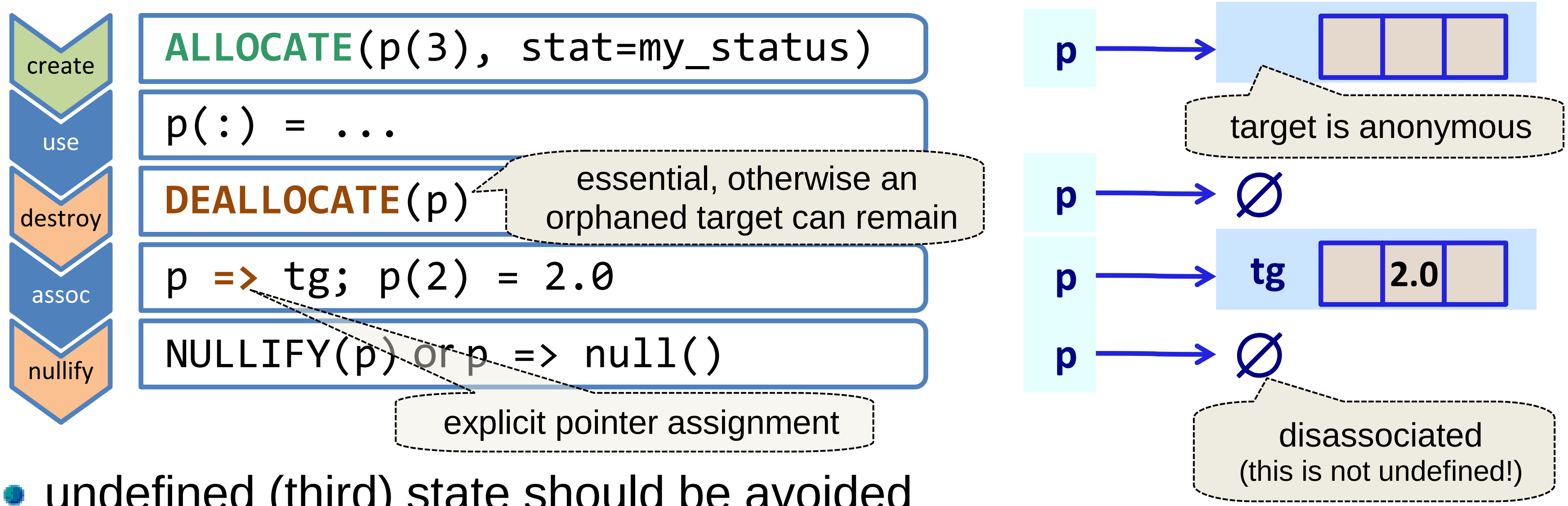
ALLOCATABLE vs. POINTER

- **An allocated allocatable entity**
  - is an object in its own right
  - becomes auto-deallocated once going out of scope

except if object has the SAVE attribute
e.g., because it is global

- **An associated pointer entity**
  - is an alias for another object, its target
  - all definitions and references are to the target

```
REAL, TARGET :: tg(3) = 0.0
```



```
ALLOCATE(p(3), stat=my_status)
p(:) = ...
DEALLOCATE(p)
p => tg; p(2) = 2.0
NULLIFY(p) or p => null()
```

essential, otherwise an orphaned target can remain

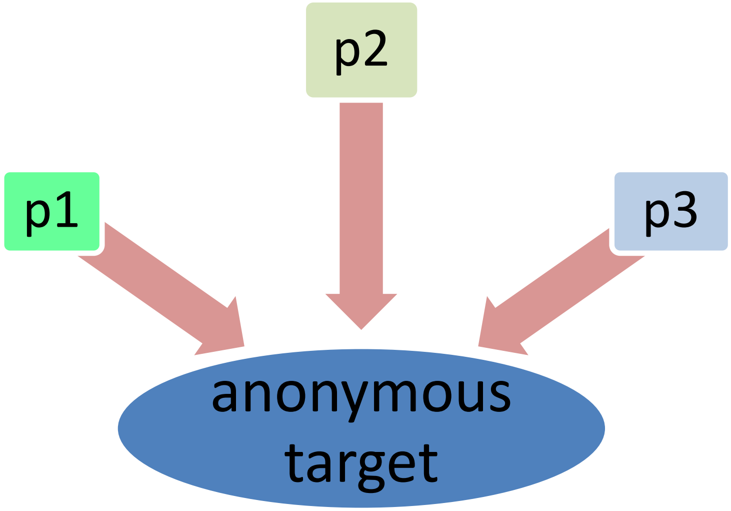
explicit pointer assignment

target is anonymous

disassociated
(this is not undefined!)

  - undefined (third) state should be avoided

Implications of POINTER aliasing



- **Multiple pointers may point to the same target**

```
ALLOCATE(p1(n))
p2 => p1; p3 => p2
```

- **Avoid dangling pointers**

```
DEALLOCATE(p2)
NULLIFY(p1, p3)
```

p2 is associated with all of the target. p1 and p3 become **undefined**

- **Not permitted: deallocation of allocatable target via a pointer**

```
REAL, ALLOCATABLE, TARGET :: t(:)
REAL, POINTER :: p(:)
```

```
ALLOCATE(t(n)); p => t
DEALLOCATE(p)
```

Features added in F03

Allocatable entities

- Scalars permitted:

```
REAL, ALLOCATABLE :: s
```

- LHS auto-(re)allocation on assignment:

conformance LHS/RHS guaranteed

```
x = p(2:m-2)
```

- The MOVE_ALLOC intrinsic:

avoid data movement

```
CALL move_alloc(from, to)
```

Pointer entities

- rank changing „=>“:

```
REAL, TARGET :: m(n)
REAL, POINTER :: p(:,:)
p(1:k1,1:k2) => m
```

rank of target must be 1

- bounds changing „=>“:

```
p(4:) => m
```

bounds remapped via lower bounds spec

Deferred-length strings:

POINTER also permitted, but subsequent use is then different

```
CHARACTER(len=:), ALLOCATABLE :: var_string

var_string = 'String of any length'
```

LHS is (re)allocated to correct length

Not a Fortran term

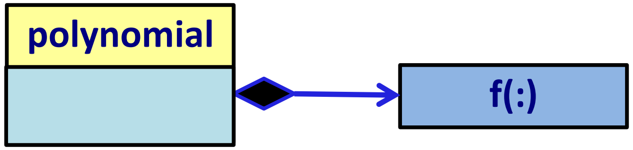
Container types

F03 Allocatable type components

```
TYPE :: polynomial
  PRIVATE
  REAL, ALLOCATABLE :: f(:)
END TYPE
```

default (initial) value is **not allocated**

- a „**value**“ container

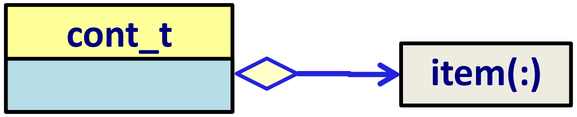


POINTER type components

```
TYPE :: cont_t
  PRIVATE
  REAL, POINTER ::
        item(:) => null()
END TYPE
```

default value is **disassociated**

- a „**reference**“ container



Container types (2):
Object declaration and assignment semantics

Allocatable type components

```
TYPE(polynomial) :: p1, p2

:
p2 = p1
```

define p1
(see e.g. next slide)

- assignment statement is equivalent to

```
IF ( ALLOCATED(p2%f) ) &
             DEALLOCATE(p2%f)
ALLOCATE (p2%f(size(p1%f)))
p2%f(:) = p1%f
```

- **„deep copy“**

POINTER type components

```
TYPE(cont_t) :: s1, s2

:
s2 = s1
```

define s1

- assignment statement is equivalent to

```
s2%item => s1%item
```

a reference,
not a copy

- **„shallow copy“**

Container types (3): Structure constructor

Allocatable type components

```
TYPE(polynomial) :: p1

p1 = polynomial( [1.0, 2.0] )
```

- dynamically allocates **p1%f** to become a size 2 array with elements 1.0 and 2.0

When object becomes undefined

- allocatable components are automatically deallocated

usually will not happen for POINTER components

POINTER type components

```
TYPE(cont_t) :: s1
REAL, TARGET :: t1(ndim)
REAL, PARAMETER :: t2(ndim) = …
```

could be omitted (default initialized component)

```
s1 = cont_t( null() )
```

- explicit target:

```
s1 = cont_t( t1 )
```

- **not** permitted:

```
s1 = cont_t( t2 )
```

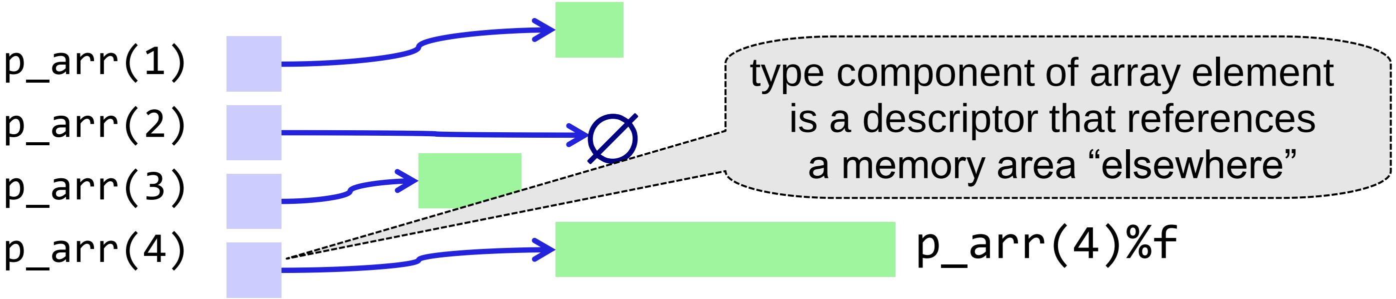
a constant cannot be a target

→ e.g., **overload** constructor to avoid this situation (create argument copy)

Container types (4): Storage layout

- **Irregularity:**
  - each array element might have a component of different length
  - or an array element might be unallocated (or disassociated)

```
TYPE(polynomial) :: p_arr(4)

p_arr(1) = polynomial( [1.0] )
p_arr(3) = polynomial( [1.0, 2.0] )
p_arr(4) = polynomial( [1.0, 2.0, 3.1, -2.1] )
```



type component of array element is a descriptor that references a memory area “elsewhere”

- **Applies for both allocatable and POINTER components**
  - a subobject designator like `p_arr(:)%f(2)` is **not** permitted

Allocatable and POINTER dummy arguments

(explicit interface required)

F03 Allocatable dummy argument

- useful for implementation of „factory procedures“ (e.g. by reading data from a file)

```
SUBROUTINE read_simulation_data(simulation_field, file_name)
  REAL, ALLOCATABLE, INTENT(OUT) :: simulation_field(:,:,:)
  CHARACTER(LEN=*), INTENT(IN) :: file_name
  :
END SUBROUTINE read_simulation_data
```

deferred-shape

implementation allocates storage after determining its size

F95 POINTER dummy argument

- example: handling of a „reference container“

```
SUBROUTINE add_reference(a_container, item)
  TYPE(cont_t) :: a_container
  REAL, POINTER, INTENT(IN) :: item(:)
  IF (associated(item)) a_container%item => item
END SUBROUTINE add_reference
```

a private pointer type component

Actual argument must have matching attribute

an exception to this exists - stay tuned

INTENT semantics for dynamic objects

specified intent	allocatable dummy object	pointer dummy object
`in`	procedure must not modify argument or change its allocation status	procedure must not change association status of object
`out`	argument becomes **deallocated** on entry auto-deallocation of `simulation_field` on previous slide!	pointer becomes **undefined** on entry
`inout`	retains allocation and definition status on entry	retains association and definition status on entry

- **„Becoming undefined“ for objects of derived type:**
  - type components become undefined if they are not default initialized
  - otherwise they get the default value from the type definition
  - allocatable type components become deallocated

INTENT(in) pointers and auto-targetting

- **Valid calls of add_reference:**

1. Actual argument has the POINTER attribute

```
TYPE(cont_t) :: my_container
REAL, POINTER :: my_item(:)
...
ALLOCATE (my_item(n))
...
CALL add_reference(my_container, my_item)
```

2. Actual argument has the TARGET attribute

F08

```
TYPE(cont_t) :: my_container
REAL, TARGET :: my_item(n)
...
CALL add_reference(my_container, my_item)
```

dummy argument is pointer associated with actual argument (“auto-targeting”)

- **Both cases require being aware of `my_item`‘s lifetime**
  - case 2 permits compile-time enforcement of actual argument‘s contiguity by adding the CONTIGUOUS attribute to the dummy argument

INTENT(OUT) and default initialized types

- **Suppose that a derived type `person` has default initialization:**

```
TYPE :: person
  CHARACTER(LEN=32) :: name = 'no_one'
  INTEGER :: age = 0
END TYPE
```

  - then, after invocation of

```
SUBROUTINE modify_person(this)
  TYPE(person), INTENT(OUT) :: this
  :
  this%name = 'Dietrich'
  ! this%age is not defined
END SUBROUTINE
```

the actual argument would have the value `person('Dietrich',0)`, i.e. components not defined inside the subprogram will be set to their default value

Quiz: what happens with a POINTER component in this situation?

Bounds of deferred-shape objects

Bounds are preserved across procedure invocations and pointer assignments

- Example:

```
REAL, POINTER :: my_item(:) => null
TYPE(cont_t) :: my_container(ndim)
ALLOCATE (my_item(-3:8))
CALL add_reference(my_container(j), my_item)
```

What arrives inside **add_reference**?

```
SUBROUTINE add_reference(…)
  :
  IF (associated(item)) a_container%item => item
```

`lbound(item)` hat the value [-3]
`ubound(item)` has the value [8]
same applies for `a_container%item`

- this is different from assumed-shape, where bounds are remapped
- it applies for both POINTER and ALLOCATABLE objects

F03 Explicit remapping of lower bounds is possible for POINTERs:

```
IF (associated(item)) a_container % item(1:) => item
```

bounds are remapped for `a_container % item`

Allocatable function results
(explicit interface required)

Scenario:

- size of function result cannot be determined at invocation
- **example:** remove duplicates from array

```
FUNCTION deduplicate(x) result(r)
  INTEGER, INTENT(IN) :: x(:)
  INTEGER, ALLOCATABLE :: r(:)
  INTEGER :: idr
  :
  ALLOCATE (r(idr))
  :
  DO i = 1, idr
    r(i) = x(…)
  END DO
END FUNCTION deduplicate
```

find number idr of distinct values

Possible invocations:

- efficient (uses auto-allocation on assignment):

```
INTEGER, ALLOCATABLE :: res(:)

res = deduplicate(array)
```

- less efficient (two function calls needed):

large enough?

```
INTEGER :: res(ndim)

res(:size(deduplicate(array))) &
      = deduplicate(array)
```

- function result is auto-deallocated after completion of invocation

It is not permitted to do
`CALL move_alloc( deduplicate(array), res )`

POINTER function results
(explicit interface required)

The POINTER attribute

- for a function result is permitted
- it is more difficult to handle on **both** the provider and the client side (need to avoid dangling pointers and potential memory leaks)

A reasonably safe example:

- extract section from container

```
FUNCTION get_section(c, s) result(r)
  TYPE(cont_t), INTENT(IN) :: c
  INTEGER, INTENT(IN) :: s(:)
  REAL, POINTER :: r(:)
  r => null()
  IF ( associated(c%item) ) &
      r => c%item(s(1):s(2):s(3))
END FUNCTION get_section
```

checks on s omitted

- no anonymous target creation involved in this case!

- invocation:

```
TYPE(cont_t) :: a_container
REAL, POINTER :: section(:)
:
section => get_section( &
 a_container, [start,end,stride] )

IF ( associated(section) ) THEN
  :
END IF
```

set up a_container

do work on section

- note the **pointer assignment**
- it is essential for implementing correct semantics and sometimes also to avoid memory leaks

Using POINTER functions in a variable definition context

F08

- **Additional permissible function invocations are:**

```
get_section(a_container, [start,end,stride] ) = x(i:j)
```

and

```
CALL array_operation(..., &
           get_section(a_container, [start,end,stride], ... )
```

corresponding dummy argument is e.g. an INTENT(inout) data argument

- **After evaluation of the function, assignment operation is performed on the result**
  - programmer needs to **guarantee** an associated pointer is returned
- **Other usage scenario: implement dictionary semantics**

```
val(weather_data, ’temperature’) = 52.8
val(weather_data, ’humidity’)    = 74
```

Opinionated recommendations

- **Dynamic entities should be used, but sparingly and systematically**
  - performance impact, avoid fragmentation of memory → allocate all needed storage at the beginning, and deallocate at the end of your program; keep allocations and deallocations properly ordered.
- **If possible, ALLOCATABLE entities should be used rather than POINTER entities**
  - avoid memory management issues (**dangling pointers** and **leaks**)
  - avoid using functions with pointer result
  - aliasing via pointers often has negative performance impact
- **A few scenarios where pointers may not be avoidable:**
  - information structures → program these in an encapsulated manner.
    The user of the facilities should normally not see a pointer at all.
  - subobject referencing (arrays and derived types) → performance impact (loss of spatial locality, suppression of vectorization)!

Recapitulation: Generic procedures

Named interfaces

```
INTERFACE generic_name
  PROCEDURE :: specific_1
  PROCEDURE :: specific_2
  …
END INTERFACE
```

- signatures of any two specifics must be sufficiently different (compile-time resolution)

Operator overloading or definition

```
INTERFACE OPERATOR (+)
  PROCEDURE :: specific_1
  PROCEDURE :: specific_2
  …
END INTERFACE
```

```
INTERFACE OPERATOR (.user_op.)
  PROCEDURE :: specific_1
  PROCEDURE :: specific_2
  …
END INTERFACE
```

Potential restrictions on signatures of specific procedures

- binary operators: functions with two arguments
- unary operators: functions with a single argument
- assignment: subroutine with two arguments
- overloaded structure constructor: function with type name as result
- user-defined derived type I/O (treated later)

Generalizing generic interface blocks

```
INTERFACE foo_generic
  MODULE PROCEDURE foo_1
  MODULE PROCEDURE foo_2
END INTERFACE
```

can be replaced by

```
INTERFACE foo_generic
  PROCEDURE foo_1
  PROCEDURE foo_2
END INTERFACE
```

with generalized functionality: Referenced procedures can be

- external procedures
- dummy procedures
- procedure pointers

Example:

```
INTERFACE foo_gen
! provide explicit interface
! for external procedure
  SUBROUTINE foo(x,n)
    REAL, INTENT(OUT) :: x
    INTEGER, INTENT(IN) :: n
  END SUBROUTINE foo
END INTERFACE
INTERFACE bar_gen
  PROCEDURE foo
END INTERFACE
```

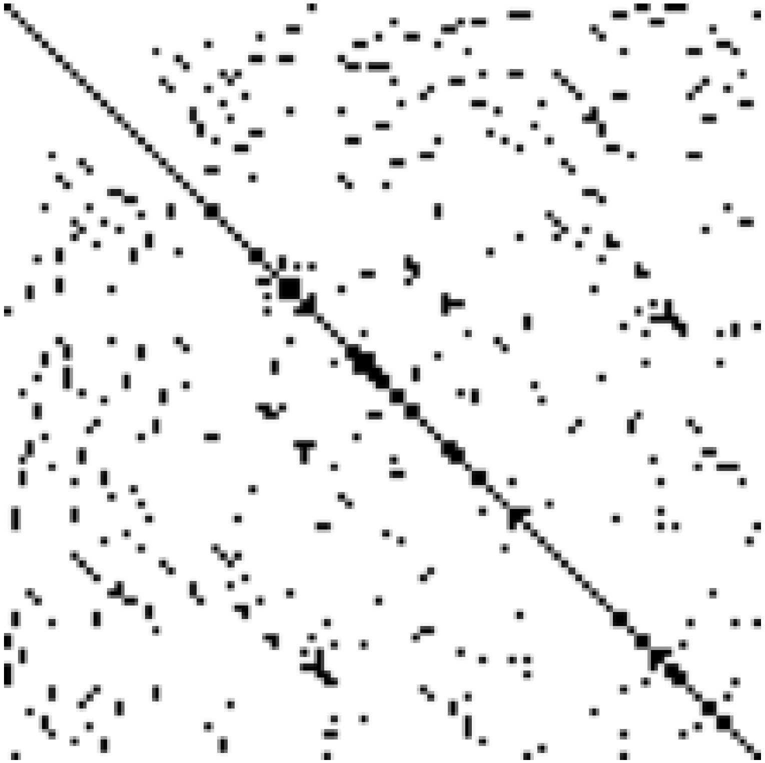
- is valid in F03
- is non-conforming if a

  `MODULE PROCEDURE`

  statement is used

Case study - sparse matrix operations

- **What is a sparse matrix?** → most entries are zero
- **Occupancy graph** → non-zero elements represented by black dots
- **Example:**

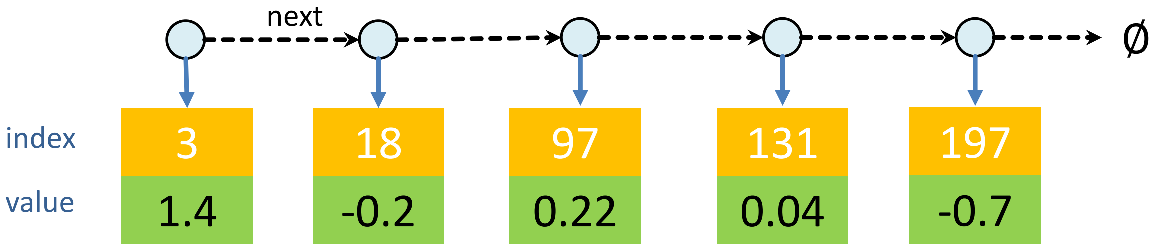


Defining a suitable data type

- **Self-referential data type (linked list)**

```
TYPE :: sparse
  PRIVATE
  INTEGER :: index
  REAL :: value
  TYPE (sparse), POINTER :: next => null()
END TYPE
```

- **A scalar object of that type can effectively hold a row of a sparse matrix:**
  - e.g., assuming a matrix dimension of 200, the 3rd row might look like

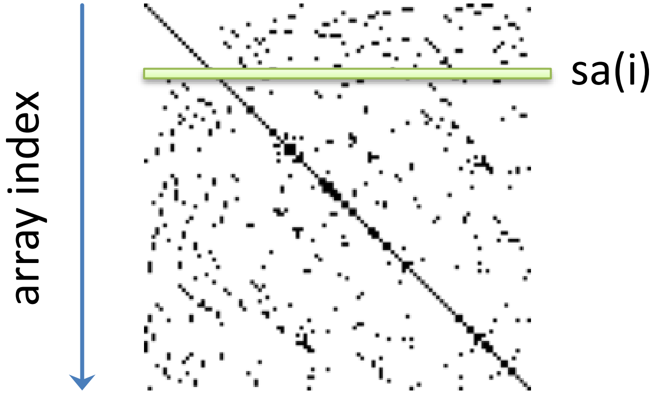


Case study - sparse matrix operations

Complete matrix

```
TYPE(sparse), ALLOCATABLE :: sa(:)
```

- **`sa(i)`** is the i-th row of the matrix
- **`sa(i)%value`** is the non-zero value of the **`sa(i)%index`** column element
- **`sa(i)%next`** is associated with the next non-zero entry



Creating, copying and operations of such objects

- topics for the next slides and the exercises

Overloading the structure constructor F03

Rationales:

- default structure constructor not generally usable due to encapsulation of type components
- default structure constructor cannot by itself set up complete list or array structures
- input data characteristics may not match requirements of default constructor

```fortran
MODULE mod_sparse
  : ! previous type definition for sparse
  INTERFACE sparse
    PROCEDURE :: create_sparse
  END INTERFACE
CONTAINS
  FUNCTION create_sparse(colidx, values) result(r)
    INTEGER, INTENT(IN) :: colidx(:)
    REAL, INTENT(IN) :: values(:)
    TYPE(sparse) :: r
    :
  END FUNCTION
END MODULE mod_sparse
```

generic has same name as the type

more than one specific is possible

must be a function with scalar result

implementation dynamically allocates the linked list for each row

Notes on overloading the structure constructor

- **If a specific overloading function has the same argument characteristics as the default structure constructor, the latter becomes unavailable**
  - advantage: for opaque types, object creation can also be done in use association contexts
  - disadvantage: it is impossible to use the overload in constant expressions

Of course, a specific may have a wildly different interface, corresponding to the desired path of creation for the object (e.g., reading it in from a file)

When default assignment is inappropriate

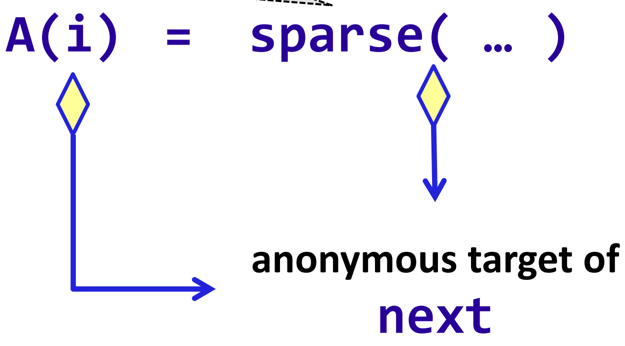
- **For the overloaded constructor, ...**

```
TYPE(sparse), ALLOCATABLE :: A(:)
:
A(i) = sparse(colidx, values)
```

allocate A

  - ... would work fine if A(i) was not previously established)

function result is discarded after assignment, but not the allocated component memory



A(i) = sparse(…)

anonymous target of **next**

- **However, for a "regular" assignment,**

```
TYPE(sparse), ALLOCATABLE :: A(:), B(:)
:
  A(i) = sparse(colidx, values)
:
B = A
```

RHS persists after the assignment

  - B effectively is not an object in its own right, but (except for the first array element in each row) links into A.

- **Also, default assignment is unavailable between objects of different derived types**

Overloading the assignment operator

- **Uses a restricted named interface:**

```
MODULE mod_sparse
  :  ! type definition of sparse
  INTERFACE assignment(=)
    PROCEDURE assign_sparse
  END INTERFACE
CONTAINS
  SUBROUTINE assign_sparse(res, src)
    TYPE(sparse), INTENT(OUT) :: res
    TYPE(sparse), INTENT(IN) :: src
    :
  END SUBROUTINE
END MODULE
```

exactly two arguments

implement a deep copy

  - create a clone of the RHS

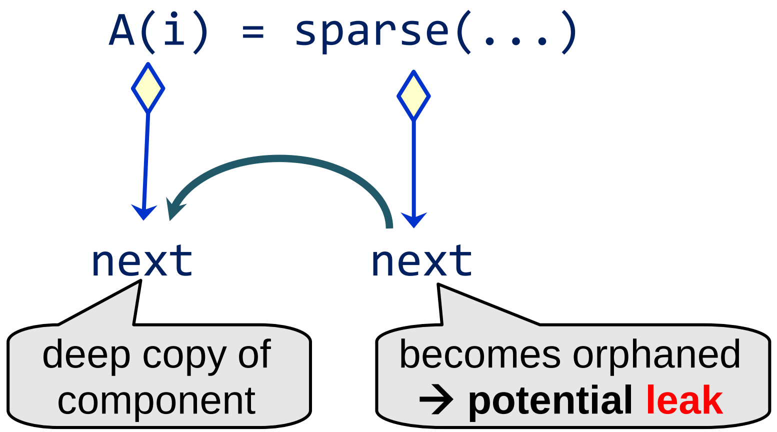
- **Further rules:**
  - first argument: **`intent(out)`** or **`intent(inout)`**
  - second argument: **`intent(in)`**
  - assignment **cannot** be overloaded for intrinsic types
  - overload usually wins out vs. intrinsic assignment.
    Exception: implicitly assigned aggregating type‘s components → aggregating type must also overload the assignment

Quiz: what might be missing in the procedure definition?

Overloaded assignment of function results: Dealing with POINTER-related memory leaks

Scenario:

- RHS may be an (overloaded) constructor or some other function value (e.g. an expression involving a defined operator)



F03 Therapy:

- add a **finalizer** to type definition
- references a module procedure with a restricted interface (usually, a single scalar argument of the type to be finalized)

```
TYPE :: sparse
  :
CONTAINS
  FINAL :: finalize_sparse
END TYPE
```

see earlier definition

Finalizing procedure implementation

applicability to array objects

```
ELEMENTAL RECURSIVE SUBROUTINE finalize_sparse(this)
  TYPE(sparse), INTENT(INOUT) :: this
  IF ( associated(this%next) ) THEN
    DEALLOCATE ( this%next )
  END IF
END SUBROUTINE
```

assumes that all targets have been dynamically allocated

Implicit execution of finalizer when ...

- object becomes undefined (e.g., goes out of scope),
- is deallocated,
- is passed to an **`intent(out)`** dummy argument, or
- appears on the left hand side of an intrinsic assignment

Quiz: what happens in the assignment
`A(i) = sparse(...)`
if a finalizer is defined, but the assignment is **not** overloaded?

Notes on finalizers

- **Feature with significant performance impact**
  - potentially large numbers of invocations:

    array elements, list members
  - finalizer invoked twice in assignments with a function value as RHS

- **Finalizers of types with pointer components:**
  - may need to consider reference counting to avoid undefined pointers
- **Non-allocatable variables in main program**
  - have the implicit SAVE attribute → are not finalized

- **Further comments on finalizers will be added later**

An alternative aliasing mechanism F03

- **Alternative: association block**
  - combine aliasing with a block construct to avoid pointer-related performance problems
- **Association syntax fragment:**

```
(<associate name> => <selector>)
```

  - allows to use the associate name as an alias for the selector inside the subsequent block
- **Very useful for**
  - heavily reused complex expressions (especially function values)
  - references into deeply nested types

- **Selector:**
  - may be a **variable** → associate name is definable
  - may be an **expression** → is pre-evaluated before aliasing to associate name, which may not be assigned to
- **Inherited properties:**
  - type, array rank and shape, polymorphism (discussed later)
  - `asynchronous`, `target` and `volatile` attributes
- **Not inherited:**
  - `pointer, allocatable` and `optional` attributes

Block construct ASSOCIATE

Example:

- given the type definitions and object declaration:

```
TYPE :: vec_3d
  REAL :: x, y, z
END TYPE
TYPE :: system
  TYPE(vec_3d) :: vec
END TYPE
TYPE :: all
  TYPE(system) :: sys
  real :: q(3)
END TYPE
TYPE(all) :: w
```

- the following block construct can be established

associate name

selector

```
ASSOCIATE( v => w%sys%vec, &
           q => sqrt(w%q) )
  v%x = v%x + q(1)**3
  v%y = v%y + q(2)**3
  v%z = v%z + q(3)**3
END ASSOCIATE
```

q must not be defined

Notes:

- more than one selector can be aliased for a single block
- the associate is auto-typed (an existing declaration in surrounding scope becomes unavailable)
- writing this out in full would be very lengthy and much less readable

following now: Exercise session 1

Object-oriented programming (I)

Type extension and polymorphism

Characterization

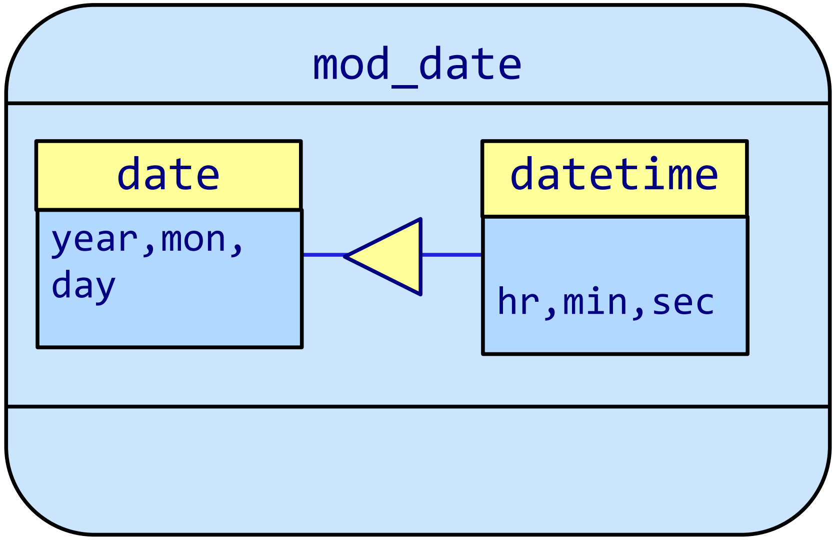
- **Terminology**
  - terms and their meaning vary between languages → danger of misunderstandings
  - will use Fortran-specific nomenclature (some commonly used terms may appear)
- **Aims of OO paradigm: improvements in**
  - re-using of existing software infrastructure
  - abstraction
  - moving from procedural to data-centric programming
  - reducing software development effort, improving productivity
- **Indiscriminate usage of OO however can be counterproductive**
  - identify “**software patterns**” which have proven useful

Scope of OO within Fortran

- **Fortran 95 supported object-based programming**
- **Today's Fortran supports object-oriented programming**
  - type extension and polymorphism (single inheritance)
  - type-bound and object-bound procedures, finalizers and type-bound generics
  - extensions to the interface concept
- **Specific intentions of Fortran object model:**
  - backward compatibility with Fortran 95
  - allow extensive correctness and consistency checking by the compiler
  - module remains the unit of encapsulation, but encapsulation becomes more fine-grained
  - design based on **Simula** object model

Type extension (1): Defining an extension

Type definitions



- idea: re-use date definition
- **datetime** a **specialization** (or **subclass**) of **date**
- **date** more general than **datetime**

Fortran type extension

```
TYPE :: date
  PRIVATE
  INTEGER :: year = 0
  INTEGER :: mon = 0, day = 0
END TYPE
TYPE, EXTENDS(date) :: datetime
  PRIVATE
  INTEGER :: hr = 0, min = 0, &
             sec = 0
END TYPE
```

- single inheritance only

Prerequisite:

- parent type must be **extensible**
- i.e., be a derived type that has neither the SEQUENCE nor the BIND(C) attribute

Type extension (2): Declaring an object of extended type

- **If type definition is public**
  - an object of the extended type can be declared in the host, or in a program unit which use associates the defining module

```
USE mod_date
:
TYPE(datetime) :: o_dt
```

- **Accessing component data**
  - **inherited** components:

    `o_dt%day   o_dt%mon   o_dt%yr`

  - **additional** components

    `o_dt%hr   o_dt%min   o_dt%sec`

- **Parent component**

  `o_dt % date`

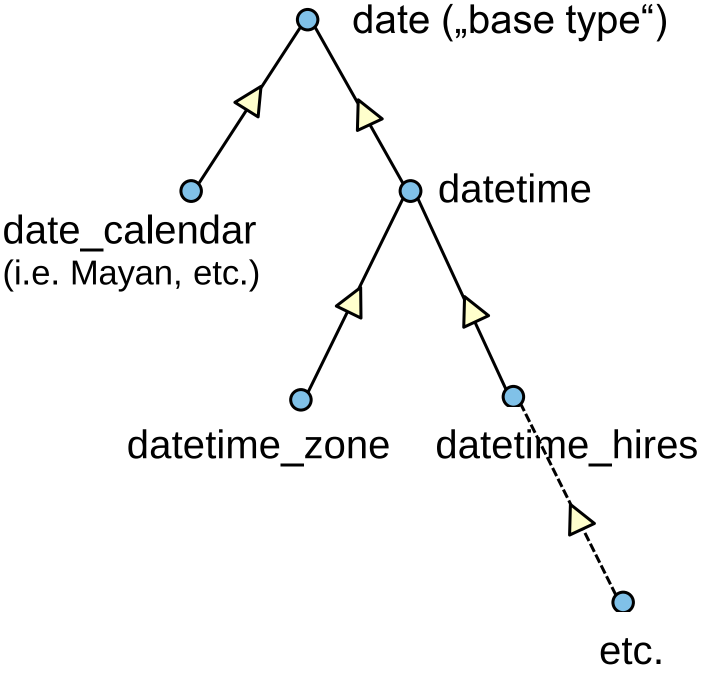
  - is an object of parent type
  - is a subobject of `o_dt`
  - recursive references possible: `o_dt % date % day`
  - parent components are themselves inherited to further extensions

- **Note:**
  - encapsulation may limit accessibility for all component variants

Type extension (3): General form of inheritance tree

A directed acyclical graph (DAG)

- this is a consequence of supporting single inheritance only



Variants:

✓ **flat** inheritance tree (typically only one level)

- base type is provided, which everyone else extends
- very often with an abstract type (discussed later) as base type

? **deep** inheritance tree

- requires care with design (which procedures are provided?) and further extension
- requires thorough documentation

Type extension (4): Further notes

- **Extension can have zero additional components**
  - use for type differentiation:

```
TYPE, EXTENDS(date) :: mydate
END TYPE
TYPE(mydate) :: o_mydate
```

  - `o_mydate` cannot be used in places where an object of `type(date)` is required
  - or to define **type-bound procedures** (discussed later) not available to parent type

- **Type parameters are also inherited**
  - see later slide for more details

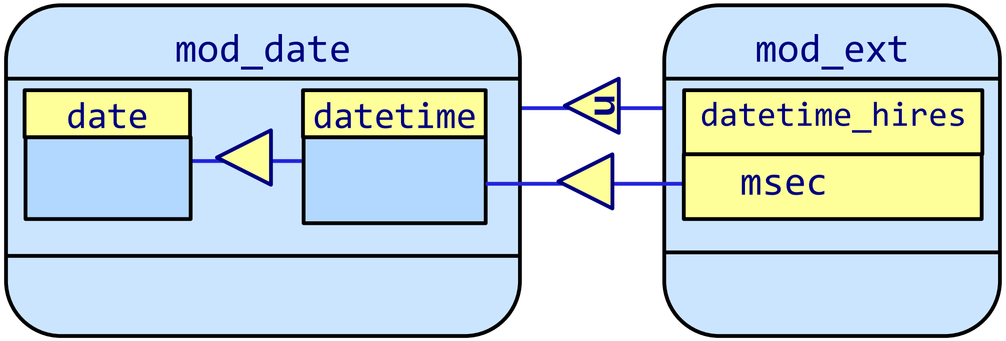
- **Inheritance and scoping:**
  - cannot have a new type component or type parameter in an extension with the same name as an inherited one

  (name space of class 2 identifiers)

Type extension (5): Component accessibility issues

Example: A type extension defined via use association

```
MODULE mod_ext
 USE mod_date
 TYPE, EXTENDS(datetime) :: &
              datetime_hires
   PUBLIC
   INTEGER :: msec
 END TYPE
 TYPE(datetime_hires) :: o_dth
END MODULE
```



Inheritance of accessibility:

- o_dth has six inherited **private** components and one **public** one
- F03 supports mixed accessibility of type components!

Technical Problem (TP1) for opaque types:

- cannot use the structure constructor for datetime_hires
- reason: it is only available outside the host of mod_date, hence **private**ness applies
- one solution: overload structure constructor

Explicit syntax for mixed component access

F03

Example: a partially opaque derived type

```
MODULE mod_person
  TYPE :: person
    PRIVATE
    CHARACTER(len=strmx) :: name
    INTEGER :: age
    CHARACTER(len=tmx), PUBLIC :: location
 END TYPE
: ! module procedures are not shown
END MODULE
```

design decision: `location` is not encapsulated. Why?

- any program unit may modify the `%location` component:

```
USE mod_person, ONLY : person
TYPE(person) :: p
: ! initialize p via an accessor defined in mod_person
p%location = 'room E.2.24'     ! update location
```

Type extension (6): Structure constructor

Using keywords

- **example:** inside the host of mod_date, one can have

```
TYPE(date) :: o_d

o_d = date (mon=9, day=12 &
            year=2012)
```

- → change component order
- rules are as for procedure keyword arguments
- e.g., once keyword use starts, it must be continued for all remaining components

Using parent component construction

- **example:** inside the host of mod_date, one can have

```
TYPE(datetime) :: o_dt

o_dt = datetime (date=o_d, &
           hr=11, min=22, sec=44)
```

- keyword notation required!

General restriction:

- it is not allowed to write overlapping definitions, or definitions that result in an incomplete object

Further structure constructor features in F03

Omitting components in the structure constructor

- this omission is only allowed for components that are **default-initialized** in the type definition
- **example:** in **any** program unit, one can have

```
USE mod_ext
TYPE (datetime_hires) :: o_hires

o_hires = datetime_hires(msec=711)
```

because all other components will receive their default-initialized value

F08

- also applies to POINTER and ALLOCATABLE components (further details on day 3)
- sometimes, this alleviates the **TP1** from some slides earlier

Polymorphism (1): Polymorphic objects

F03

Declaration with CLASS:

```
CLASS(date), ... :: o_poly_dt
```

possible additional attributes

- **declared** type is date
- **dynamic** type may vary at run time: may be declared type and all its (known) extensions (type compatibility)

loosening of strict F95 typing rules

- direct access (i.e., references and definitions) only possible to components of **declared** type (compile-time: compiler lacks knowledge, run-time: semantic problem and performance issues)

Data item can be

1. dummy data object

   interface polymorphism

2. pointer or allocatable variable

   data polymorphism →
   a new kind of dynamic entity

3. both of the above

```
o_poly_dt%day = 12
```
✓

```
o_poly_dt%hr = 7
```

invalid even if dynamic type of `o_poly_dt` is `datetime`

Polymorphism (2): Interface polymorphism

Example:

- increment date object by a given number of days

```
SUBROUTINE inc_date(this, days)
  CLASS(date), INTENT(INOUT) :: this
  REAL(RK), INTENT(IN) :: days
  : ! implementation → exercise
END SUBROUTINE
```

Inheritance mechanism: actual argument ...

- … can be of declared type of dummy or an extension:

```
TYPE(date) :: o1
TYPE(datetime) :: o2
: ! initialize both objects
CALL inc_date(o1,2._rk)
CALL inc_date(o2,2._rk)
```

could replace „TYPE(…)“ by „CLASS(…)“ for both objects
(an additional attribute may be needed)

- ... can be polymorphic or non-polymorphic

Argument association:

- **dynamic** type of actual argument is assumed by the dummy argument

Polymorphism (3): Interface polymorphism cont'd

Example continued:

- account for fraction of a day when incrementing a `datetime` object

```
SUBROUTINE inc_datetime(this, days)
  CLASS(datetime), &
            INTENT(INOUT) :: this
  REAL(rk), INTENT(IN) :: days
  : ! implementation → exercise
END SUBROUTINE
```

Restriction on use:

- cannot take objects of declared type `date` as actual argument:

```
CLASS(date) :: o1
CLASS(datetime) :: o2
: ! initialize both objects

CALL inc_datetime(o1,.03_rk)

CALL inc_datetime(o2,.03_rk)
```

assume dummy arguments

invalid invocation – will not compile (this also applies if `o1` is of non-polymorphic `type(date)`)

- **reason:** if `o1` has dynamic type `date`, then no `sec` component exists that can be incremented

Fortran term:

- dummy argument must be **type compatible** with actual argument

  (note that type compatibility, in general, is **not** a symmetric relation)

Polymorphism (4): Data polymorphism / dynamic objects

Declaration:

```
CLASS(date), ALLOCATABLE :: ad
```

polymorphic allocatable scalar

```
CLASS(date), ALLOCATABLE :: bd(:)
```

polymorphic allocatable array

```
CLASS(date), POINTER :: &
                    cd => null()
```

polymorphic pointer to scalar

```
: ! etc
```

- unallocated / disassociated entities: dynamic type is equal to declared type
- usual difference in semantics (e.g., auto-deallocation for allocatables)

Producing valid entities:

- **typed** allocation to base type or an extension

```
ALLOCATE(datetime :: ad, cd)
```

becomes dynamic type

```
ALLOCATE(date :: bd(5))
```

could omit since equal to base type

- pointer association

```
TYPE(datetime_zone), &
             target :: t
…
! may need to deallocate cd

cd => t
```

dynamic type of `cd` is now `datetime_zone`

Polymorphism (5): Arrays

- **A polymorphic object may be an array**

```
CLASS(date) :: ar_d(:)
```

  - here: assumed-shape

    (Note: using assumed-size or explicit-shape is usually not a good idea)

but type information applies for all array elements

  - all array elements have the **same** dynamic type

- **For per-element type variation:**
  - define an array of suitably defined derived type:

```
TYPE :: date_container
  CLASS(date), ALLOCATABLE :: p
END TYPE

TYPE(date_container) :: ard(10)
```

  - `ard(1)%p` can have a dynamic type different from that of `ard(2)%p`

Polymorphism (6): Further allocation mechanisms

object ad: declared two slides earlier

Sourced allocation

- produce a **clone** of a variable or expression

```
CLASS(datetime) :: src
: ! define src
ALLOCATE(ad, source=src)
```

- allocated variable (ad) must be type compatible with source
- source can, but need not be polymorphic
- definition of dynamic type of source may be inaccessible in the executing program unit (!)
- usual semantics: deep copy for allocatable components, shallow copy for pointer components

Sourced allocation of arrays

F08 array bounds are also transferred in sourced allocation

Molded allocation F08

- allocate an entity with the same shape, type and type parameters as mold

```
CLASS(datetime) :: b

ALLOCATE(ad, mold=b)
```

- mold need not have a defined value (no data are transferred)
- otherwise, comparable rules as for sourced allocation

Polymorphism (7): Type resolution

Example scenario:

- a routine is needed that writes a complete object of CLASS(date) to a file irrespective of its dynamic type

```fortran
SUBROUTINE write_date(this, fname)
  CLASS(date), intent(in) :: this
  CHARACTER(len=*) :: fname
  : ! open file fname on unit
  : ! see inset right
END SUBROUTINE
```

Problem:

- how can extended type components be accessed within write_date?

New block construct:

must be polymorphic

```fortran
SELECT TYPE (this)
TYPE IS (date)
  WRITE(unit,fmt='(“date“)')
  WRITE(unit,…) this%day,…
TYPE IS (datetime)
  WRITE(unit,fmt='(“datetime“)')
  WRITE(unit,…) …,this%hr,…


: ! further type guards for
: ! other extensions
CLASS DEFAULT
  STOP 'Type not recognized'
END SELECT
```

inside this **type guard** block:
- this is nonpolymorphic
- type of this is datetime

fall-through block:
- this is polymorphic
- typically used for error processing

Polymorphism (8): Semantics and rules for SELECT TYPE

- **Execution sequence:**
  - at most one block is executed
  - selection of block:

1. find **type guard** („type is“) that exactly matches the dynamic type
2. if none exists, select **class guard** („class is“) which most closely matches dynamic type and is still type compatible

   → **at most one such guard exists**
3. if none exists, execute block of **class default** (if it exists)

- **Access to components**
  - in accordance with resolved type (or class)

- **Resolved polymorphic object**
  - must be type compatible with every type/class guard (constraint on guard!)
- **Technical problem (TP2):**
  - access to all extension types' definitions is needed to completely cover the inheritance tree
- **Type selection allows both**
  - run time type identification (**RTTI**)
  - run time class identification (**RTCI**)

It is necessary to ensure type safety and (reasonably) good performance

- RTCI or mixed RTTI+RTCI are not expected to occur very often
- executing SELECT TYPE is an expensive operation

An RTCI scenario

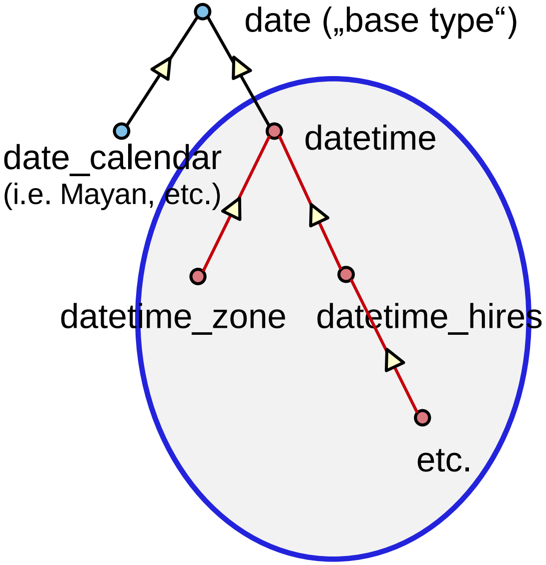
„Lifting“ to an extended type

- e.g., because a procedure must be executed which only works (polymorphically or otherwise) for the extended type
- remember invalid invocation of inc_datetime from earlier slide – we can now write a viable version of this:

```
CLASS(date) :: o1
: ! initialize o1

SELECT TYPE (o1)
CLASS IS (datetime)
  CALL inc_datetime(o1,.03_rk)

CLASS DEFAULT
  WRITE(*,*) &
      'Cannot invoke inc_datetime on o1'
END SELECT
```

inside „class is“ block:

- **o1** is polymorphic (this is what we want here!)
- declared type of **o1** is datetime



part of inheritance tree covered by class guard

SELECT TYPE and association

- **Associated alias must be used if the selector is not a named variable**
  - e.g., if it is a type component, or an expression
- **Additional restrictions:**
  - only one selector may appear
  - the selector must be polymorphic
- **Example:**
  - given the type definition

```
TYPE :: person
  CLASS(date), ALLOCATABLE :: birthday
END TYPE
```

and an object o_p of that type, the RTTI for o_p%birthday is **required** to look like this:

```
SELECT TYPE ( b => &
              o_p%birthday )
CLASS IS (date)
  WRITE(*,*) 'Birthday:', &
            b%day, b%mon, b%year
CLASS IS (datetime)
  …
  WRITE(*,*) 'Birth hour:', b%hr
END SELECT
```

Polymorphism (9): A universal base class

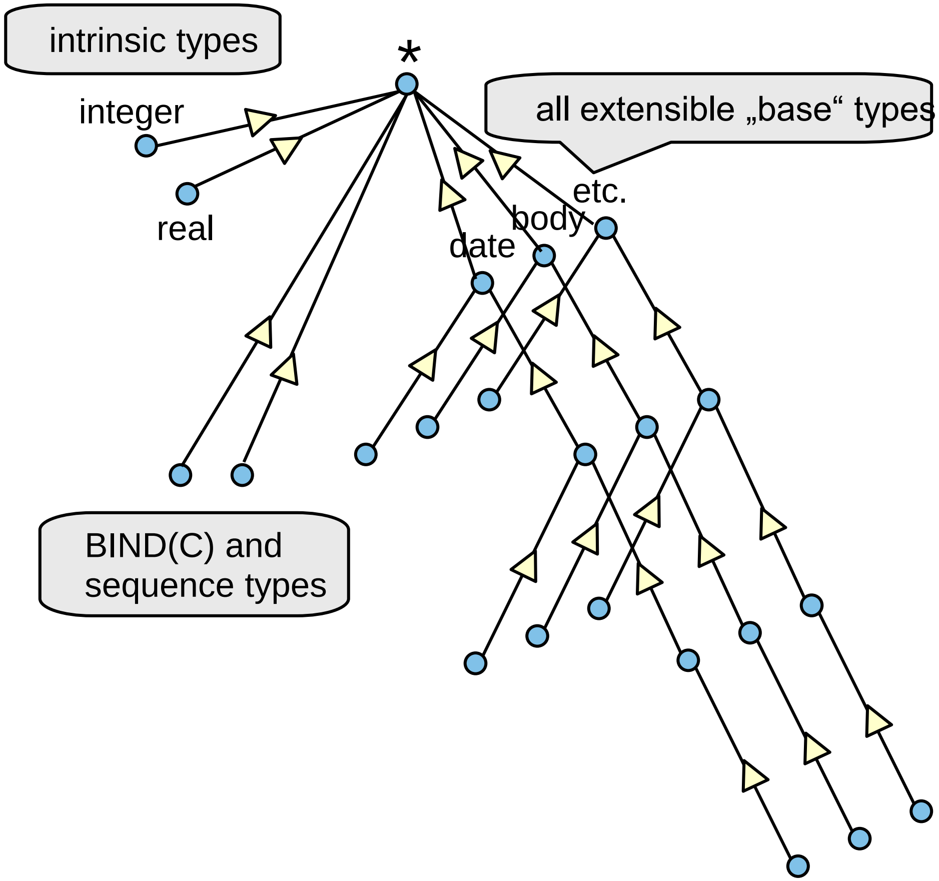
- **Denoted as „*“**
  - „no declared type“
- **Refers to an object that is of**
  1. intrinsic, or
  2. extensible, or
  3. non-extensible

  dynamic type
- **Syntax:**

```
CLASS(*), ... :: o_up
```

- **an unlimited polymorphic (UP) entity**
  - usual restrictions: (POINTER eor ALLOCATABLE) or a dummy argument, or both

- **Conceptual inheritance tree:**



Polymorphism (10): UP pointer

- **An UP pointer can point to anything:**

```
CLASS(*), POINTER :: p_up
TYPE(datetime), TARGET :: o_dt
REAL, POINTER :: rval

p_up => o_dt
ALLOCATE(rval) ; rval = 3.0
p_up => rval
```

- **However, dereferencing …**

```
p_up => o_dt
WRITE(*, *) p_up % yr
! will not compile
```

… is not allowed without a SELECT TYPE block (no declared type → no accessible components)

```
TYPE(datetime), POINTER :: pt

SELECT TYPE (p_up)
TYPE IS (datetime)
  WRITE(*, *) p_up % yr
  pt => p_up
TYPE IS (real)
  WRITE(*, '(f12.5)') p_up
CLASS DEFAULT
  WRITE(*, *) 'unknown type'
END SELECT
```

- **RTTI:**
  - can also use an **intrinsic** type guard in this context
  - analogous for UP dummy arguments if access to data is needed

UP entities of non-extensible dynamic type

- **Use of this form of UP is not recommended**
  - Reason: different from intrinsic and extensible types, **no** type information is available via the object itself → SELECT TYPE always falls through to „class default“
- **Loss of type safety:**
  - syntactically, it is in this case allowed to have

```
CLASS(*), TARGET :: o_up
TYPE(...), POINTER :: p_nonext

p_nonext => o_up
```

of **arbitrary** dynamic type

any BIND(C) or SEQUENCE type

  - use this feature only if you know what you‘re doing
    (i.e. maintain type information separately and **always** check)

See **`examples/day2/discriminated_union`** for a possible usage scenario

Polymorphism (11): Allocating an UP object

- **Applies to**
  - unlimited polymorphic entities with the POINTER or ALLOCATABLE attribute
- **Typed allocation:**
  - any type may be specified, including intrinsic and non-extensible types
- **Sourced or molded allocation**
  - `source` or `mold` may be of any type (limitation to extensible type does not apply)
  - the newly created object takes on the dynamic type of `source` or `mold` (same as for „regular“ polymorphic objects)

Polymorphism (12): Type inquiry intrinsics

- **Compare dynamic types:**

```
extends_type_of(a, mold)

same_type_as(a, b)
```

.TRUE. if `mold` is type compatible with `a`

  - functions return a logical value
  - arguments must be entities of extensible (dynamic) type, which
  - can be polymorphic or non-polymorphic

- **Recommendation:**

it may be implicitly available!

  - only use if type information is not available (most typically if at least one of the arguments is UP), or if type information not relevant for the executed algorithm

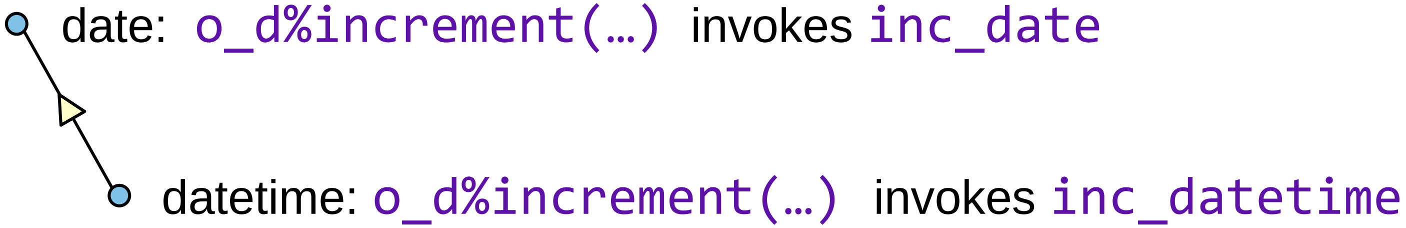
Object-oriented programming (II)

Binding of procedures to Types and Objects

Motivation

- **Remember inc_date and inc_datetime procedures:**
  - programmer decides which of the two routines is invoked
  - for an object of dynamic type date, inc_datetime cannot be invoked
- **Suppose there is a desire to**
  - invoke incrementation depending on the **dynamic** type of the object:

```
CLASS(date), ALLOCATABLE :: o_d
```



date: o_d%increment(…) invokes inc_date

datetime: o_d%increment(…) invokes inc_datetime

- **This concept is also known as dynamic (single) dispatch via the object**

not a Fortran term

  - cannot use F95 style generics (polymorphism forces run-time decision)

Prolegomenon: Pointers to procedures (1)

F03

Declaration:

```
PROCEDURE(subr), POINTER :: &
                 pr => null()
```

- a named procedure pointer with an explicit interface ...
- … here it is:

```
INTERFACE
  SUBROUTINE subr(x)
    REAL, INTENT(INOUT) :: x
  END SUBROUTINE
END INTERFACE
```

Usage:

```
REAL :: x
:
pr => subr
x = 3.0
CALL pr(x) ! invokes "subr"
```

must associate **before** invocation

Notes:

- pointing at a procedure that is defined with a generic or elemental interface is not allowed
- no TARGET attribute is required for the procedure pointed to

Pointers to procedures (2)

- **Functions are also allowed in this context:**

```
INTERFACE
  REAL FUNCTION fun(x)
    REAL, INTENT(IN) :: x
  END FUNCTION
END INTERFACE

PROCEDURE(fun), POINTER :: &
                  pfun => null()
```

- **Usage:**

```
pfun => fun

WRITE(*,*) pfun(3.5)

pfun => sin

WRITE(*,*) pfun(3.0)
```

returns fun(3.5)

returns sin(3.0)

- this also illustrates that the target can change throughout execution (in this case to the intrinsic `sin`)
- some of the intrinsics get dispensation for being used like this despite being generic

Pointers to procedures (3)

Using an implicit interface ⚠

- not recommended (no signature checking, many restrictions)

```
PROCEDURE(), POINTER :: pi => null()
EXTERNAL :: targ_1, targ_2

! external, pointer :: pi => null()

PROCEDURE(), POINTER :: pfi
REAL :: pfi, targ_2
```

equivalent alternative

type declaration for **pfi** indicates a function pointer

- invocations:

```
pi => targ_1
CALL pi(x, y, z)  ! OK if consistent with interface

pi => pfun        !  💣  target has explicit interface

pfi => targ_2        ! OK if interface + function result
WRITE(*,*) pfi(x, y) ! consistent
```

not permitted

Procedures as type components

F03

- **Two variants are supported:**

object-bound procedure (OBP) and **type-bound procedure (TBP)**

```
TYPE :: data_send_container
  CLASS(data), ALLOCATABLE :: d
  PROCEDURE(send), &
    POINTER :: send => null()
END TYPE
```

good practice, but not obligatory

- **Syntax:**
  - „standard“ type component
  - pointer to a procedure
- **Semantics:**
  - each object's `%send` component can be associated with any procedure with the same interface as `send`

```
TYPE :: date
   : ! previously defined comp.
CONTAINS
  PROCEDURE :: &
         increment => inc_date
END TYPE
```

existing procedure

- **Syntax:**
  - component in `contains` part of type definition
  - **no** POINTER attribute appears
- **Semantics:**
  - each object's `%increment` component is associated with the procedure `inc_date`

Restrictions on the procedure interface

... apply for both variants

This is the dummy that will usually become argument associated with the object invoking the TBP

- **First dummy argument:**
  - declared type must be same type as the **type (type of the object) the procedure is bound to (the procedure pointer is a component of)**
  - must be polymorphic if and only if type is extensible (→ assure inheritance works with respect to any invocation)
  - must be a scalar
  - must not have the POINTER or ALLOCATABLE attribute

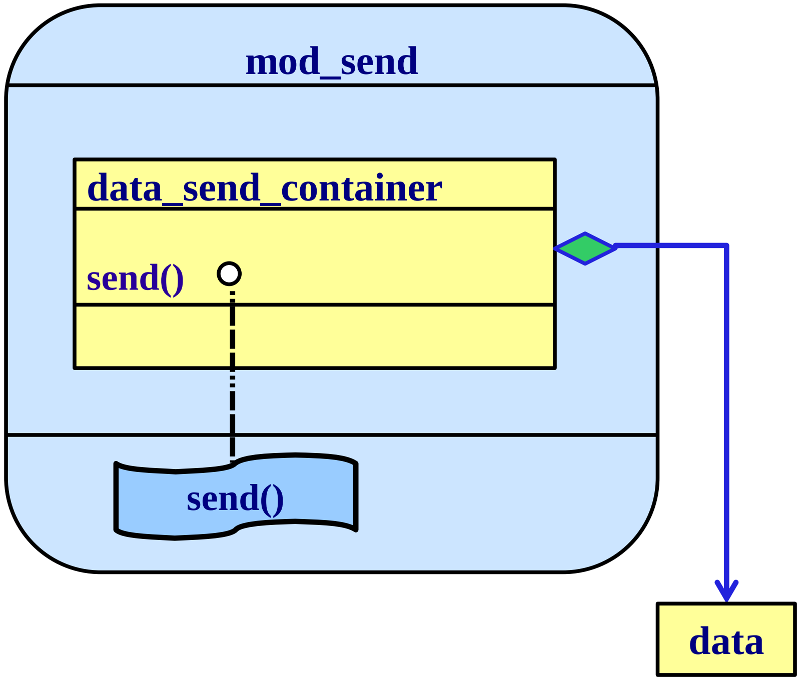
```
SUBROUTINE send(this, desc)
  CLASS (data_send_container) :: this
  CLASS (handle) :: desc
  : ! implementation not shown
END SUBROUTINE
```

object-bound case

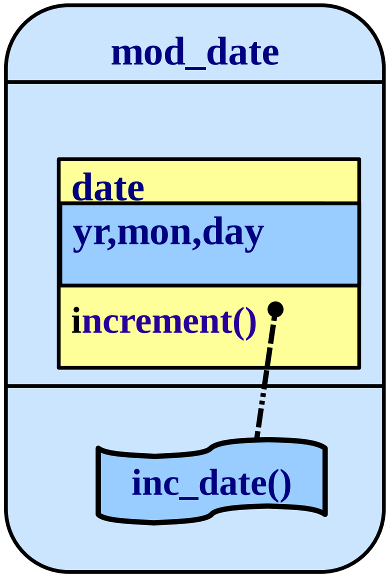
- for the type-bound case, the procedure interface has already been specified on an earlier slide

Diagrammatic representation

- **Object-bound procedure**



- **Type-bound procedure (TBP)**



  - implementation need not be public
  - `increment` component is **public** (even if type is opaque), unless explicitly declared private

Invocation of procedure components

- **Syntax is the same for the object-bound and type-bound case**
  - need to set up pointer association for the object-bound case before invocation

```
TYPE (data_send_container) :: c
: ! set up desc
ALLOCATE (c%d, source = …)
IF (…) THEN
 c%send => my_send1
ELSE
 c%send => my_send2
END IF

CALL c%send(desc)
```

same interface as **`send`**

object-bound case

assume first **`if`** branch is taken →
same as **`call my_send1(c, desc)`**

```
TYPE (date) :: o_d
TYPE (datetime) :: o_dt

o_d = date(12, 'Dec', 2012)
: ! also make o_dt defined

CALL o_d%increment(12._rk)
```

type-bound case

same as **`call inc_date(o_d, 12._rk)`**

- **Notes:**
  - the object is associated **with the first dummy** of the invoked procedure („**passed object**“)
  - **inheritance:**

```
CALL o_dt%increment(2._rk)
```

(as things stand now) also invokes `inc_date`, so we haven't yet gotten what we wanted some slides earlier

Overriding a type-bound procedure

In a type extension,

- an existing accessible TBP can be **overridden**:

```
TYPE, EXTENDS(date) :: datetime
  : ! as before
CONTAINS
  PROCEDURE :: increment => &
                     inc_datetime
END TYPE
```

with the binding above added,

```
CALL o_dt%increment(.03_rk)
```

invokes `inc_datetime`

Invoke by type component

- a class 2 name → no name space collisions between differently typed objects (with or without inheritance relation)

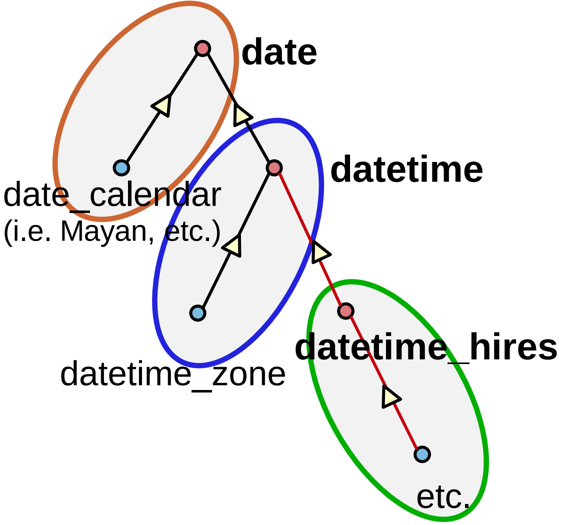
Invoking object: may be polymorphic or not polym.

- **dynamic** type is used to decide which procedure is invoked
- this procedure is **unique**: go up the inheritance tree until a binding is found (implicit RTCI)

Assumption:

Bold-faced types define or override TBP `increment`

Others don't



- type may be **inaccessible** in invocation's scope!

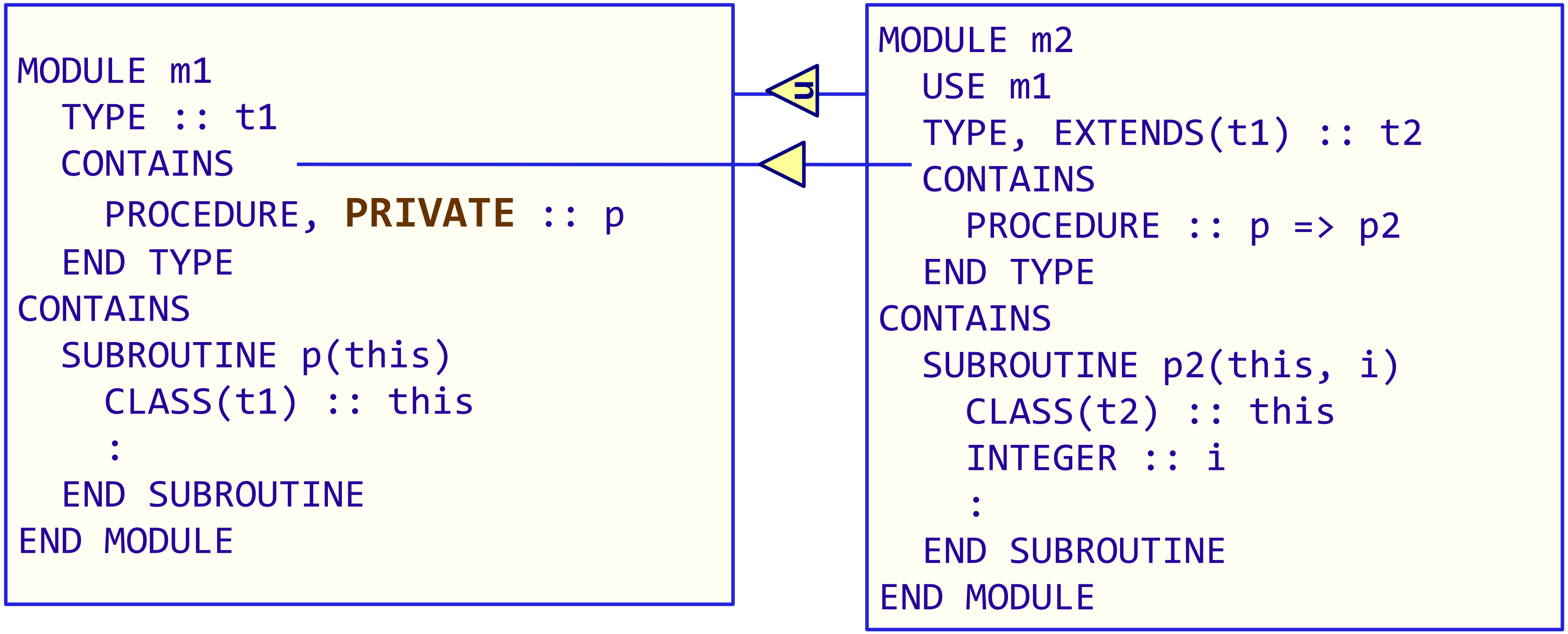
Restrictions on the interface of a procedure used for overriding an existing TBP

- **Each must have same interface as the original TBP**
  - even same argument keyword names!
  - if they (both!) are functions, the result characteristics must be the same
- **Except the passed object dummy,**
  - which must be declared `class(`*`<extended type>`*`)`
- **This guarantees that inheritance works correctly together with dynamic dispatch**
- **In the `datetime` example,**
  - the procedure interface of `inc_datetime` (see earlier slide) obeys these rules

Comment on private type-bound procedures

A F08 Corrigendum

- **These cannot be overridden outside their defining module**



```
MODULE m1
  TYPE :: t1
  CONTAINS
    PROCEDURE, PRIVATE :: p
  END TYPE
CONTAINS
  SUBROUTINE p(this)
    CLASS(t1) :: this
    :
  END SUBROUTINE
END MODULE
```

```
MODULE m2
  USE m1
  TYPE, EXTENDS(t1) :: t2
  CONTAINS
    PROCEDURE :: p => p2
  END TYPE
CONTAINS
  SUBROUTINE p2(this, i)
    CLASS(t2) :: this
    INTEGER :: i
    :
  END SUBROUTINE
END MODULE
```

- therefore **p2** is not an overriding type-bound procedure, but a **new binding** that applies to all entities of **`CLASS(t2)`**
- **p2** therefore needs not to have the same characteristics as **p**

Note: compilers might get dynamic dispatch wrong in this situation, and don't handle differing interfaces (check recent releases)

Suppress overriding in extension types

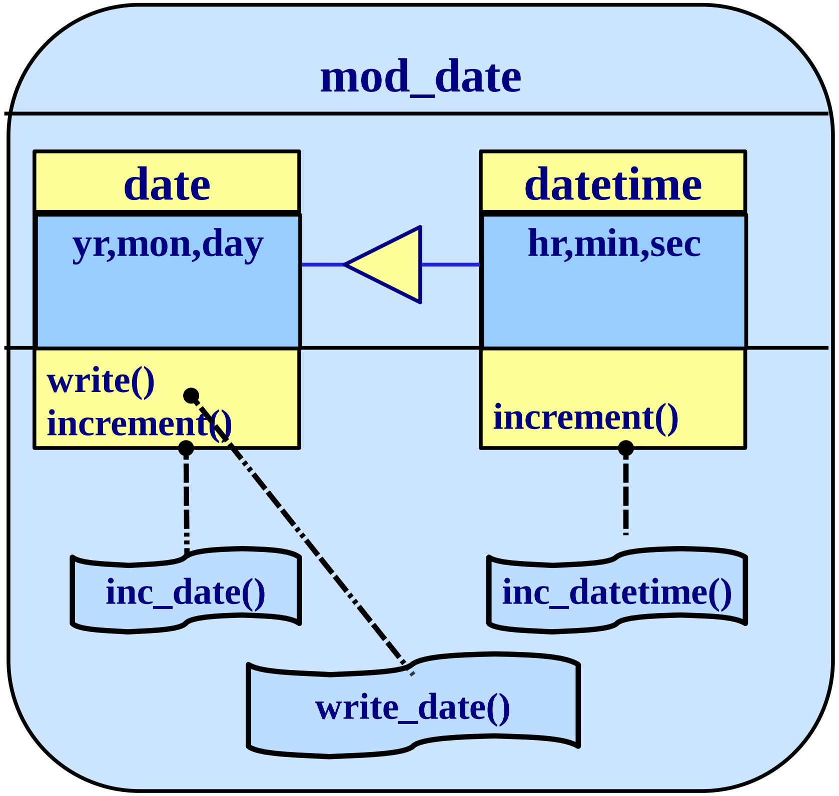
- **The NON_OVERRIDABLE attribute can be used in any binding**

- **For example, if `write_date`** (see earlier slide) **is bound to `date` as follows:**

```
TYPE :: date
  : ! previously defined comp.
CONTAINS
  PROCEDURE :: increment => inc_date
  PROCEDURE, NON_OVERRIDABLE :: write => write_date
END TYPE
```

  - then it is not possible to override the `write` TBP in any extension
  - this makes sense here because it is intended that the complete inheritance tree is dealt with inside the implementation of the procedure (other rationales may exist in other scenarios)

Diagrammatic representation for overriding TBPs



Non-overridden procedures are inherited

On „SELECT TYPE“ vs. „overriding TBP”

- **Dynamic dispatch by TBP**
  - TBP's should behave **consistently** whether handed an entity of base type or any of its extensions (Liskov substitution principle)
  - example: “incrementation by (fractional) days“ obeys the substitution principle
  - some attention is needed to avoid violations:
    - client extends a type
    - programmer using the interface may misinterpret intended semantics (→ documentation issue!)
  - avoid bad design of extensions (analogous to side effects in functions)
  - **Example:** derive square from rectangle (exercise)

```
TYPE(datetime) :: dtt
CALL dtt%date%increment(120._rk)
```

- **Isolate RTTI**
  - to the few places where needed
    - creation of objects, I/O
  - since it is all too easy to forget covering all parts of the inheritance tree
- **RTCI rarely used, because TBPs fill that role**
- **Overriding does not lose functionality**
  - parent type invocation (see left)

Array as passed object

- **Passed object must be a scalar**
  - therefore, arrays must usually invoke TBP or OBP elementwise
- **But a type-bound procedure may be declared ELEMENTAL**
  - actual argument then may be an array
    (remember further restrictions on interface of an ELEMENTAL procedure)
  - invocation can be done with array or array slice

```
TYPE :: elt
  :
CONTAINS
  PROCEDURE :: p
END TYPE
```

```
ELEMENTAL SUBROUTINE p(this, x)
  CLASS(elt), INTENT(INOUT) :: this
  REAL, INTENT(IN) :: x
  : ! no side effects
END SUBROUTINE
```

```
TYPE(elt) :: o(5)
:
CALL o%p([ (real(i), i=1,5) ])
```

invocation

- **This is not feasible for the object-bound case**
  - each elements' procedure pointer component may point to a different procedure

Variations on the passed object: PASS and NOPASS

Pass non-first argument

- via explicit keyword specification
- **example:** bind procedure to more than one type

```
TYPE :: t1
  :
CONTAINS
  PROCEDURE, &
  PASS(o1) :: pf
END TYPE
```

no „=>“. Why?

```
TYPE :: t2
  :
CONTAINS
  PROCEDURE, &
  PASS(o2) :: &
       pq => pf
END TYPE
```

```
SUBROUTINE pf(o1, x, o2, y)
  CLASS(t1) :: o1
  CLASS(t2) :: o2
  :
END SUBROUTINE
```

Do not pass argument at all

```
TYPE :: t3
  :
CONTAINS
  PROCEDURE, NOPASS :: pf
END TYPE
```

Invocations:

```
TYPE(t1) :: o_t1
TYPE(t2) :: o_t2
TYPE(t3) :: o_t3
:
CALL o_t1%pf(x, o_t2, y)
CALL o_t2%pq(o_t1, x, y)
CALL o_t3%pf(o_t1, x, o_t2, y)
```

Note:

- overriding TBPs must preserve PASS / NOPASS

Abstract Types

F03

Properties:

- no entity of that (dynamic) type can exist
- may have zero or more components

```
TYPE, ABSTRACT :: <type name>
  : ! components, if any
[ CONTAINS
  : ! type-bound procedures
]
END TYPE
```

- declaration of a polymorphic entity of declared abstract type is permitted
- an abstract type may be an extension

Example:

```
TYPE, ABSTRACT :: shape
END TYPE

TYPE, EXTENDS(shape) :: square
  REAL :: side
END TYPE
```

- valid and invalid usage:

```
TYPE(shape) :: s1          💣
TYPE(square) :: s2          ✓
CLASS(shape), ALLOCATABLE :: &
                s3, s4         ✓

ALLOCATE(shape :: s3)        💣
ALLOCATE(square :: s4)        ✓
```

Abstract Types with deferred TBPs (aka Interface Classes)

F03

Syntax of definition

- one or more deferred bindings are added:

```
TYPE, ABSTRACT :: handle
  PRIVATE
  INTEGER :: state = 0
CONTAINS
  PROCEDURE(open_handle), &
    DEFERRED :: open

  PROCEDURE, &
   NON_OVERRIDABLE :: getstate
END TYPE HANDLE
```

only allowed in an abstract type

- cannot override a non-deferred binding with a deferred one

Deferred binding:

- described by an interface (usually abstract)

```
ABSTRACT INTERFACE
  SUBROUTINE open_handle(this, &
                         info)
    IMPORT :: handle
    CLASS(handle) :: this
    CLASS(*), INTENT(IN), &
          OPTIONAL :: info
  END SUBROUTINE
END INTERFACE
```

assuming type definition in host

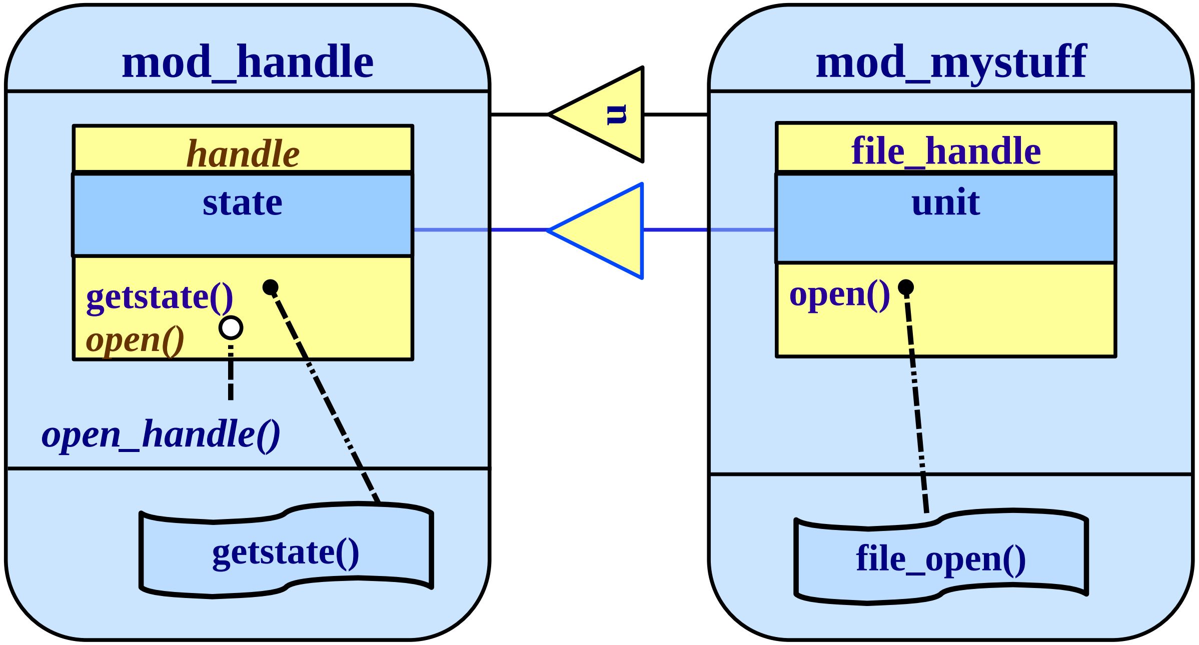
- enforces that any client defining a type extension **must** establish an overriding binding (once you have one, it is inherited to extensions of the extension)

Extending from an interface class

```
MODULE mod_file_handle
  USE mod_handle
  TYPE, EXTENDS(handle) :: file_handle
    PRIVATE
    INTEGER :: unit
  CONTAINS
    PROCEDURE :: open => file_open
  END TYPE file_handle
CONTAINS
  SUBROUTINE file_open(this, info)
    CLASS(file_handle) :: this
    CLASS(*), INTENT(IN), OPTIONAL :: info
    SELECT TYPE (info)
    TYPE IS (character(len=*))
      : ! open file with name info and store this%unit
      this%state = 1
    : ! error handling via class default
    END SELECT
  END SUBROUTINE
END MODULE mod_file_handle
```

will not compile without this override

Diagrammatic representation of the interface class and its realization



- **Will typically use (at least) two separate modules**
  - e.g., module providing abstract type often third-party-provided
- **Abstract class and abstract interface indicated by italics**
  - non-overridable TBP getstate() → “invariant method”

Using the interface class

```
PROGRAM prog_client
  USE mod_file_handle, ONLY : handle, file_handle
  IMPLICIT NONE

  CLASS(handle), ALLOCATABLE :: h
  ALLOCATE(file_handle :: h)
  CALL h%open('output_file.dat')
  : ! further processing including I/O
  : ! close file
  DEALLOCATE(h)
END PROGRAM prog_client
```

full dependency inversion would imply that use association is only to **`mod_handle`**

Compare to „traditional“ design:

- **Implementation details of non-abstract type decoupled from “policy-based” design of abstract type**
- **Dependency inversion:**
  - ideally, both clients and implementations depend on abstractions
  - in a procedural design, the type “handle” would need to contain all possible variants → abstraction becomes dependent on irrelevant details

Dependency Inversion with Submodules

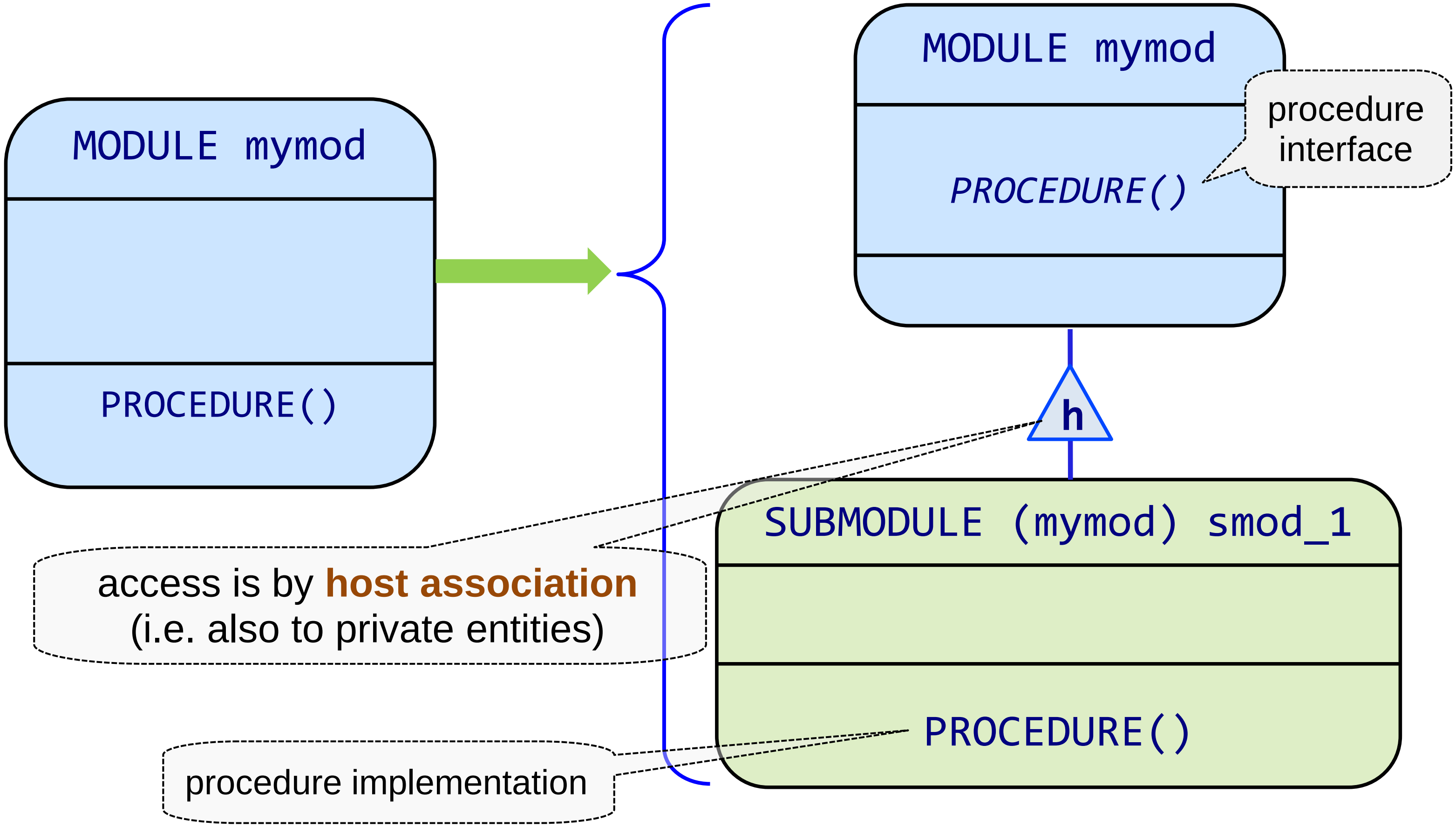
Problems with Modules

- **Tendency towards monster modules for large projects**
  - e.g., type component privatization prevents programmer from breaking up modules where needed
- **Recompilation cascade effect**
  - changes to module procedures forces recompilation of all code that use associates that module, even if specifications and interfaces are unchanged
  - workarounds are available, but somewhat clunky
- **Object oriented programming**
  - more situations with potential circular module dependencies are possible (remember **TP2** on earlier slide)
  - type definitions referencing each other may also occur in object-based programming

Solution: Submodules

F08

- **Split off implementations (module procedures) into separate files**



Submodule program units

Syntax

ancestor module

```
SUBMODULE ( mymod ) smod_1
  : ! specifications
CONTAINS
  : ! implementations
END SUBMODULE
```

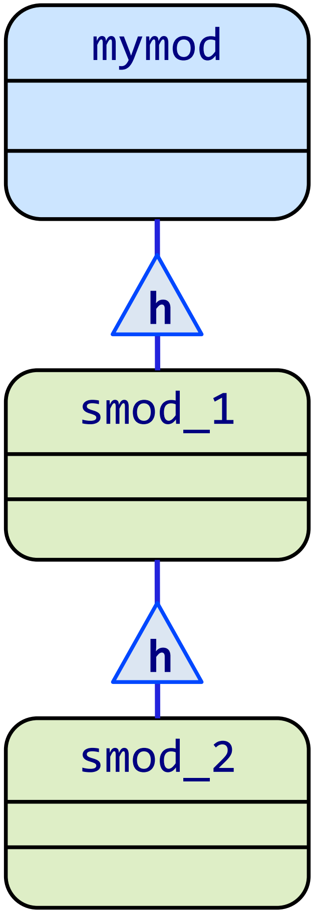
- applies recursively: a descendant of **smod_1** is

```
SUBMODULE ( mymod:smod_1 ) smod_2
  :
END SUBMODULE
```

immediate ancestor submodule

- sibling submodules are permitted (but avoid duplicates for accessible procedures)

Symbolic representation



Submodule specification part

- **Like that of a module, except**
  - no **PRIVATE** or **PUBLIC** statement or attribute can appear
- **Reason: all entities are private**
  - and only visible inside the submodule and its descendants

```
MODULE mymod
  IMPLICIT NONE
  TYPE :: t
    :
  END TYPE
:
END MODULE
```

```
SUBMODULE ( mymod ) smod_1
  TYPE, EXTENDS(t) :: ts
    :
  END TYPE
  REAL, ALLOCATABLE :: x(:,:)
:
END SUBMODULE
```

effectively private

Separate module procedure interface

In specification part of the ancestor module

```
MODULE mod_date
 TYPE :: date
  : ! as previously defined
 END TYPE
 INTERFACE
  MODULE SUBROUTINE write_date (this, fname)
   CLASS(date), INTENT(IN) :: this
   CHARACTER(LEN=*), INTENT(IN) :: fname
  END SUBROUTINE
  MODULE FUNCTION create_date (year, mon, day) result(dt)
   INTEGER, INTENT(IN) :: year, mon, day
   TYPE(date) :: dt
  END FUNCTION
 END INTERFACE
END MODULE
```

indication that the implementation is contained in a submodule

- `IMPORT` statement not permitted (auto-import is done)

Separate module procedure implementation

- **Variant 1:**
  - complete interface (including argument keywords) is taken from module
  - dummy argument and function result declarations are not needed

```
SUBMODULE (mod_date) date_procedures
  : ! specification part
CONTAINS
  MODULE PROCEDURE write_date
    : ! local variable-decls and executable
    : ! statements as shown before
  END PROCEDURE write_date
  MODULE PROCEDURE create_date
    : ! local variable-decls and executable
    : ! statements as shown before
  END PROCEDURE create_date
END SUBMODULE date_procedures
```

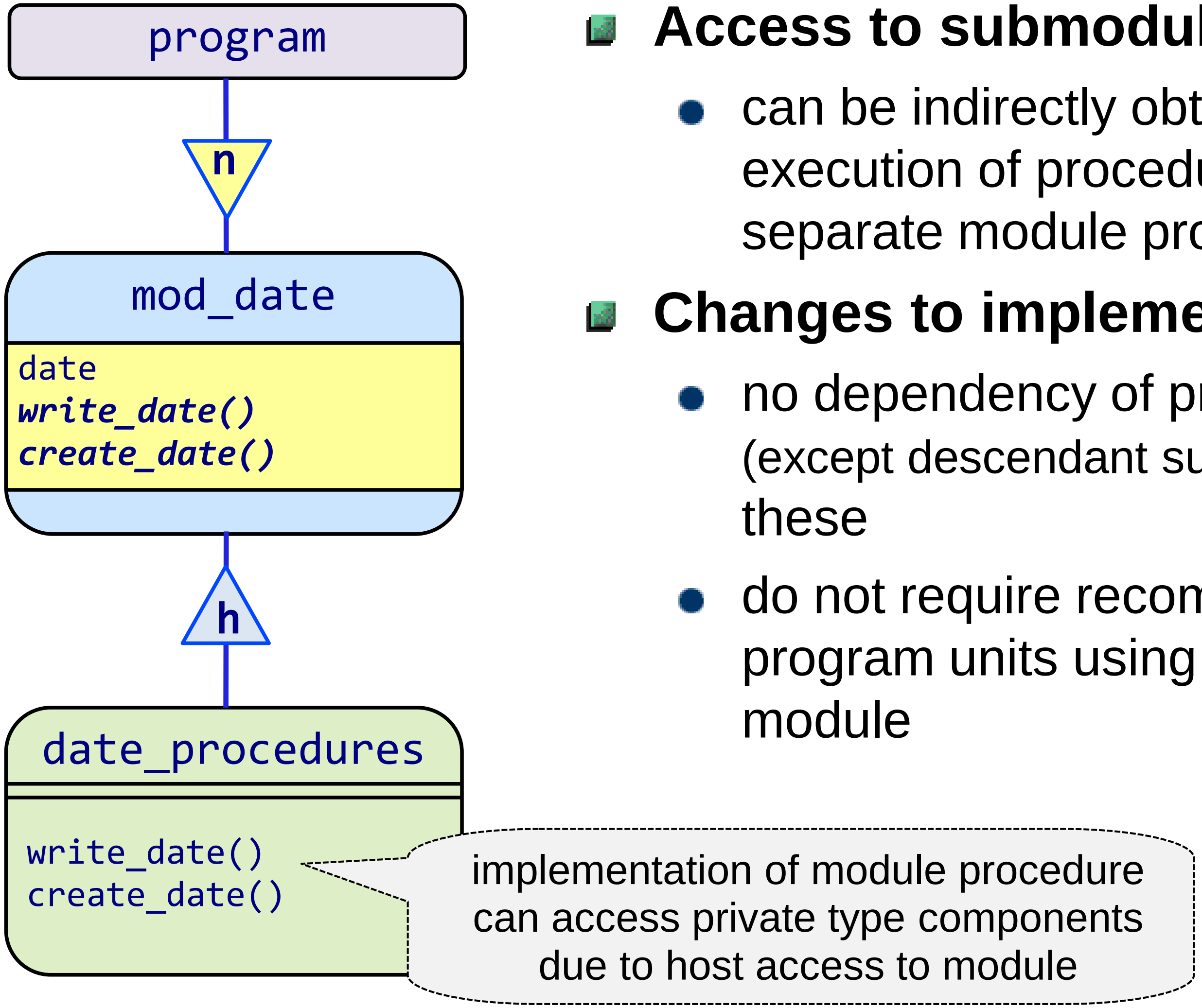
Separate module procedure implementation

- **Variant 2:**
  - interface is replicated in the submodule
  - must be consistent with ancestor specification

```
SUBMODULE (mod_date) date_procedures
  : ! specification part
CONTAINS
  MODULE SUBROUTINE write_date (this, fname)
      CLASS(date), INTENT(IN) :: this
      CHARACTER(LEN=*), INTENT(IN) :: fname
      : ! local variable-decls and executable
      : ! statements as shown before
  END SUBROUTINE write_date
  MODULE FUNCTION create_date (year, mon, day) result(dt)
      INTEGER, INTENT(IN) :: year, mon, day
      TYPE(DATE) :: dt
    : ! ... as shown before
  END FUNCTION create_date
END SUBMODULE date_procedures
```

note syntactic difference to Variant 1

Dependency inversion explained



implementation of module procedure can access private type components due to host access to module

Access to submodule entities

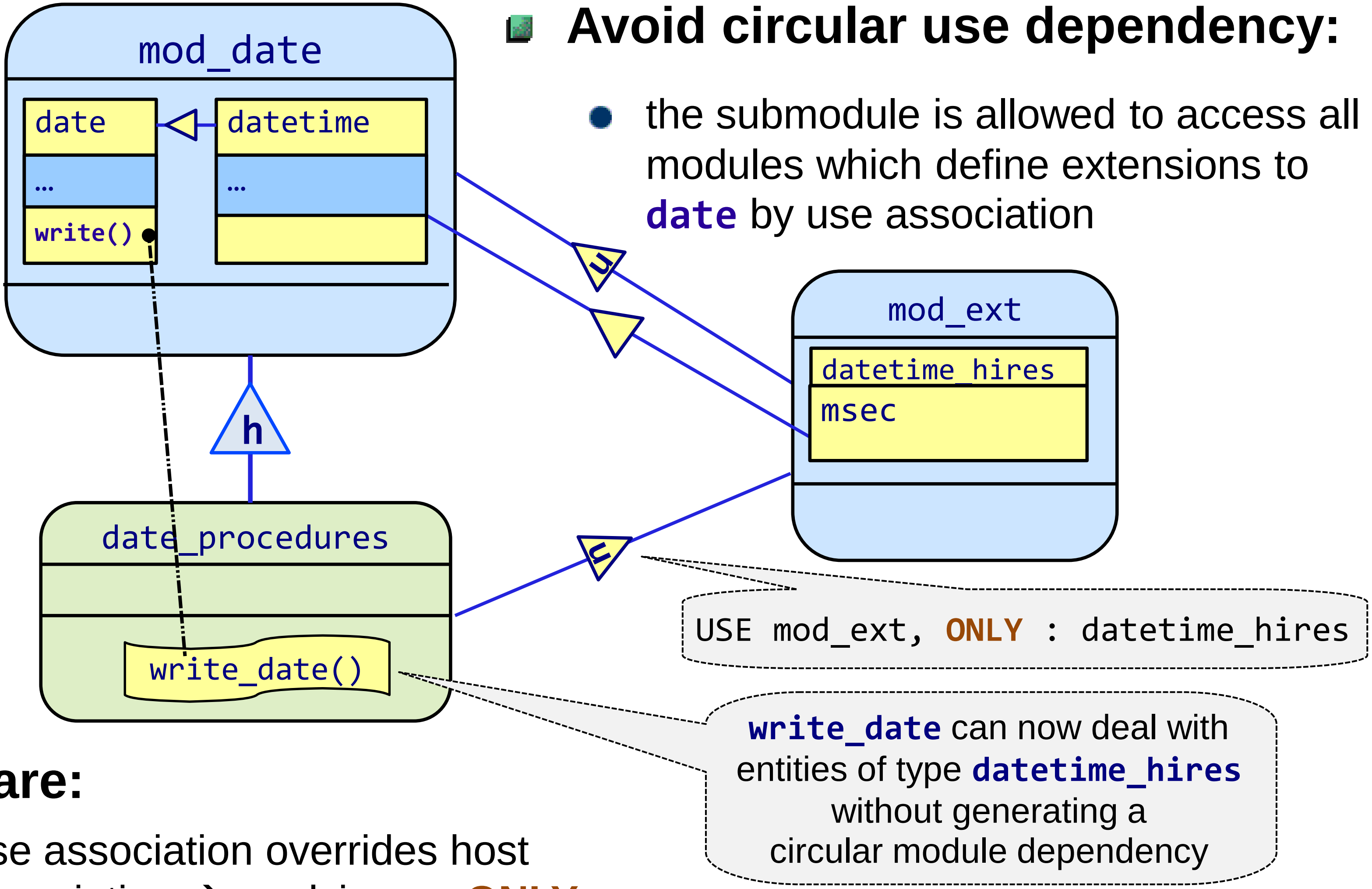
- can be indirectly obtained via execution of procedures declared with separate module procedure interfaces

Changes to implementations

- no dependency of program units (except descendant submodules) on these
- do not require recompilation of program units using the parent module

Exploiting dependency inversion in OO design

- **Avoid circular use dependency:**
  - the submodule is allowed to access all modules which define extensions to **`date`** by use association



```
USE mod_ext, ONLY : datetime_hires
```

`write_date` can now deal with entities of type **`datetime_hires`** without generating a circular module dependency

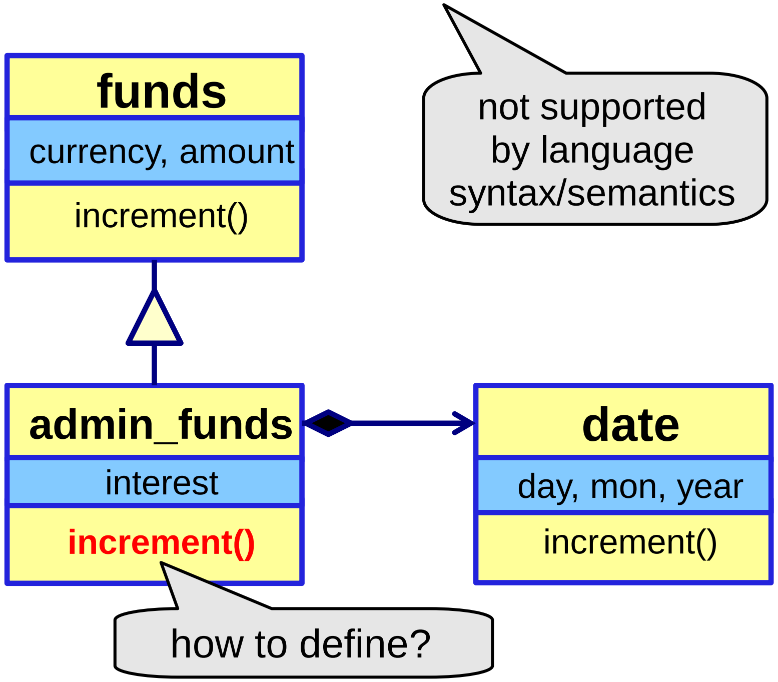
- **Beware:**
  - use association overrides host association → applying an **ONLY** clause is advisable

following now: Exercise session 2

Generic Type-bound Procedures

Example scenario

- **Two existing concepts**
  - both support an interface of same name and function
- **Need to join those concepts**
  - which may interact in some way
  - scenario: multiple inheritance



- **TBP `increment()`:**
  - for `funds`, increments amount
  - for `date`, increments by days
  - for `admin_funds`, **both** the above should work individually, and in addition it should be possible to account for the interest rate (interaction!)
- **These are interfaces with differing signatures!**
  - in principle, the `funds` binding will be inherited by `admin_funds`
  - remember interface restrictions on overriding a TBP

Declaring a generic type-bound procedure

Starting point:

- the type which first declares the binding that must be generic

```
TYPE, PUBLIC :: funds
  PRIVATE
  CHARACTER(len=3) :: currency
  REAL :: amount
CONTAINS

  PROCEDURE, PRIVATE :: &
                    inc_funds
  GENERIC, PUBLIC :: &
       increment => inc_funds
END TYPE
```

good manners
to hide this

OCP

- may need to retrofit generic from simple TBP (easily done, at the cost of recompiling all clients)

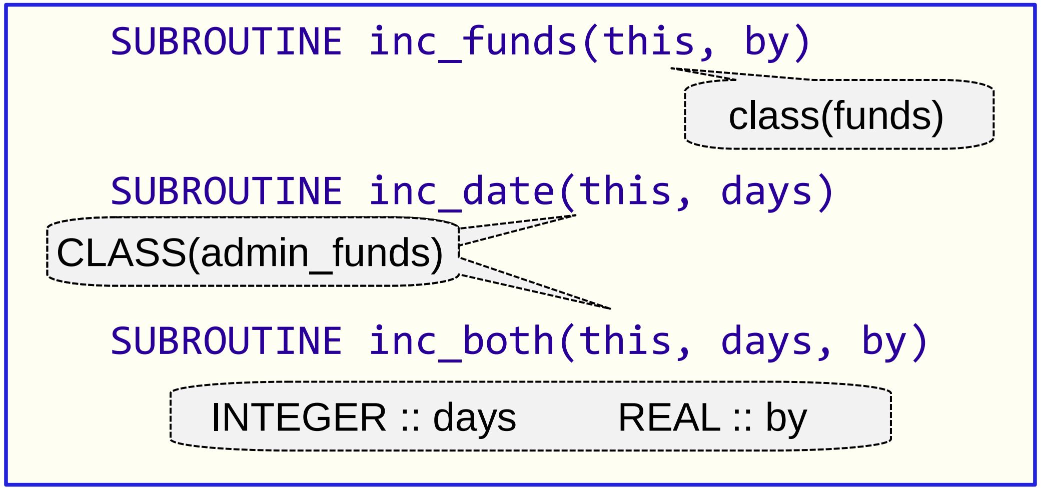
Adding specifics to a generic in a type extension:

```
TYPE, PUBLIC, &
   EXTENDS(funds) :: admin_funds
  PRIVATE
  REAL :: interest
  TYPE(date) :: d
CONTAINS
  PROCEDURE, PRIVATE :: inc_date
  PROCEDURE, PUBLIC :: inc_both
  GENERIC, PUBLIC :: &
  increment => inc_date, inc_both
END TYPE
```

aggregation

- three specific TBPs now can be invoked via one generic name (one inherited, two added)
- it is also allowed to bind to an inherited specific TBP

Disambiguating procedure interfaces



```
   SUBROUTINE inc_funds(this, by)

   SUBROUTINE inc_date(this, days)

   SUBROUTINE inc_both(this, days, by)
```

class(funds)

CLASS(admin_funds)

INTEGER :: days   REAL :: by

Implementation ...

… is inherited

… re-dispatches to `this%d%increment()`

… invokes both the above, after accounting for interest

Selection of specific TBP:

- must be possible at compile time
- pre-requisite: between each pair of specifics, for at least one non-optional argument type incompatibility is required

providing two specifics which only differ in one argument, one being type compatible with the other, is not sufficient to disambiguate

Invocation of a generic TBP

```
TYPE(admin_funds) :: of
CLASS(funds), &
      allocatable :: of_poly

ALLOCATE(admin_funds :: of_poly)
: ! initialize both objects

CALL of%increment(12, 600.)

CALL of%increment(17)

CALL of%increment(100.)

CALL of_poly%increment(1, 2.)
```

how can this be fixed?

- **The usual TKR (type/kind/rank) matching rules apply …**

Compile-time resolution ...

… to **inc_both()**

… to **inc_date()**

… to **inc_funds()**

… is not possible because this interface is not defined for an entity of declared type **funds**

- a specific TBP can still be overridden i.e., compile-time resolution is only partial

See **examples/multiple_inheritance**

Overriding a specific binding in a generic TBP

- **Further type extension (in a different module)**

```
TYPE, EXTENDS(admin_funds) :: &
              my_funds
  :
CONTAINS
  PROCEDURE :: &
        inc_both => inc_my_funds
END TYPE
```

original binding **public** so it can be overridden

- with a module procedure:

```
SUBROUTINE inc_my_funds(this, &
               ninc, by)
  CLASS(my_funds) :: this
  : ! ninc, by as before
END SUBROUTINE
```

- **Invocation:**

```
CLASS(admin_funds), &
      ALLOCATABLE :: o_mf

ALLOCATE(my_funds :: o_mf)
: ! initialize o_mf

CALL o_mf%increment(1, 23.)
```

invokes overriding procedure **inc_my_funds** because dynamic type is **my_funds**

Unnamed generic TBPs – defined operator

Example:

- unary trace operator

```
TYPE, PUBLIC :: matrix
  PRIVATE
  REAL, ALLOCATABLE :: element(:,:)
CONTAINS
  PROCEDURE, PUBLIC :: trace
  GENERIC, PUBLIC :: &
      OPERATOR(.tr.) => trace
END TYPE matrix
```

- the NOPASS attribute is not allowed for unnamed generics

```
REAL FUNCTION trace(this)
  CLASS(matrix), INTENT(IN) :: this
  :
END FUNCTION
```

Invocation:

```
TYPE (matrix) :: o_mat
: ! initialize object
WRITE(*,*) 'Trace of o_mat is ',&
           .tr. o_mat
```

Rules and restrictions:

- same rules and restrictions (e.g., with respect to characteristics) as for generic interfaces and their module procedures
- **here:** procedure must be a function with an INTENT(IN) argument

Note:

- inheritance → statically typed function result may be insufficient

Unnamed generic TBPs – overloaded operator

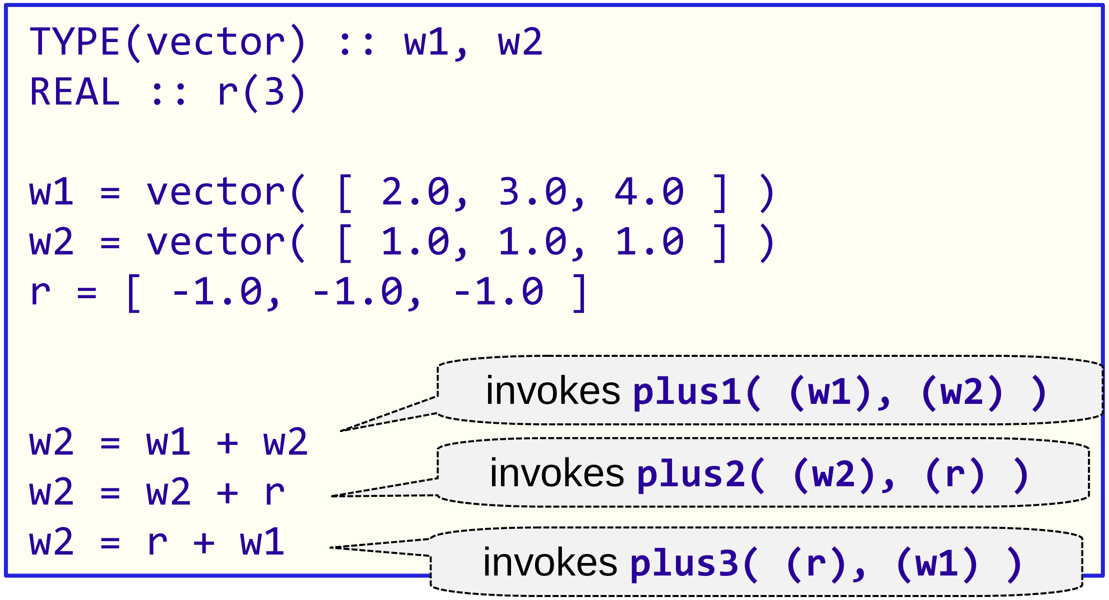
- **Overloading allowed for**
  - existing operators
  - assignment
- **Example:**

```
TYPE :: vector
  : ! see earlier definition
CONTAINS
  PROCEDURE :: plus1, plus2
  PROCEDURE, PASS(v2) :: plus3
  GENERIC, PUBLIC :: OPERATOR(+) => &
                     plus1, plus2, plus3
END TYPE matrix
```

- **Specifics:**

```
FUNCTION plus1(v1, v2)
  CLASS(vector), INTENT(IN) :: v1
  TYPE(vector), INTENT(IN) :: v2
  TYPE(vector) :: plus1
  : ! implementation omitted
END FUNCTION
FUNCTION plus2(v1, r)
  CLASS(vector), INTENT(IN) :: v1
  REAL, INTENT(IN) :: r(:)
  TYPE(vector) :: plus2
  : ! implementation omitted
END FUNCTION
FUNCTION plus3(r, v2)
CLASS(vector), INTENT(IN) :: v2
  REAL, INTENT(IN) :: r(:)
  TYPE(vector) :: plus3
  : ! implementation omitted
END FUNCTION
```

Using the overloaded operator



```
TYPE(vector) :: w1, w2
REAL :: r(3)

w1 = vector( [ 2.0, 3.0, 4.0 ] )
w2 = vector( [ 1.0, 1.0, 1.0 ] )
r = [ -1.0, -1.0, -1.0 ]


w2 = w1 + w2
w2 = w2 + r
w2 = r + w1
```

invokes **`plus1( (w1), (w2) )`**

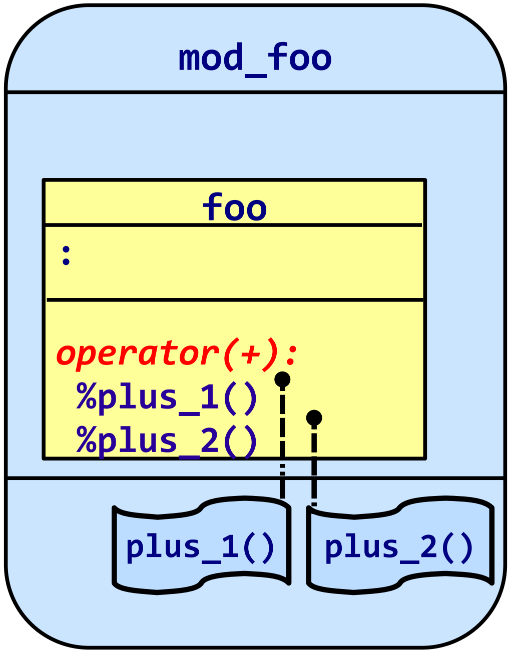
invokes **`plus2( (w2), (r) )`**

invokes **`plus3( (r), (w1) )`**

Remaining problem:

- how to deal with polymorphism –
- for an extension of **`vector`**, the result usually should also be of the extended type
- but: function result must be declared consistently for an override

Diagrammatic representation of generic TBPs



Use italics to indicate generic-ness

- provide list of specific TBPs as usual
- overriding in subclasses can then be indicated as previously shown

Advanced I/O Topics

Reminder on error handling for I/O

- **An I/O statement may fail:**
  Examples:
  - opening a non-existing file with status=‘OLD‘
  - reading beyond the end of a file
- Without additional measures:
  RTL will terminate the program
- Prevent termination via:
  user-defined error handling
  - specify an `iostat` and possibly `iomsg` argument in the I/O statement
  - use of `err` / `end` / `eor = <label>` is also possible but is legacy!

  → **do not use in new code!!**

- **`iostat=ios` specification**

  `ios` (scalar default integer) will be:
  - negative — if end of file detected,
  - positive — if an error occurs,
  - zero — otherwise

- **`iomsg=errstr` specification**

  `errstr` (default character string of sufficient length) supplied with appropriate description of the error if `iostat` is none-zero

- **Use intrinsic logical functions:**

```
is_iostat_end(ios)
is_iostat_eor(ios)
```

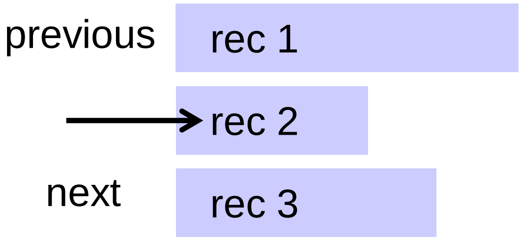
to check iostat-value of I/O operation

for EOF (end of file) or EOR (end of record)

condition

Nonadvancing I/O (1)

Allow file position to vary inside a record:



previous rec 1

rec 2

next rec 3

Syntactic support:

- ADVANCE specifier in formatted READ or WRITE statement

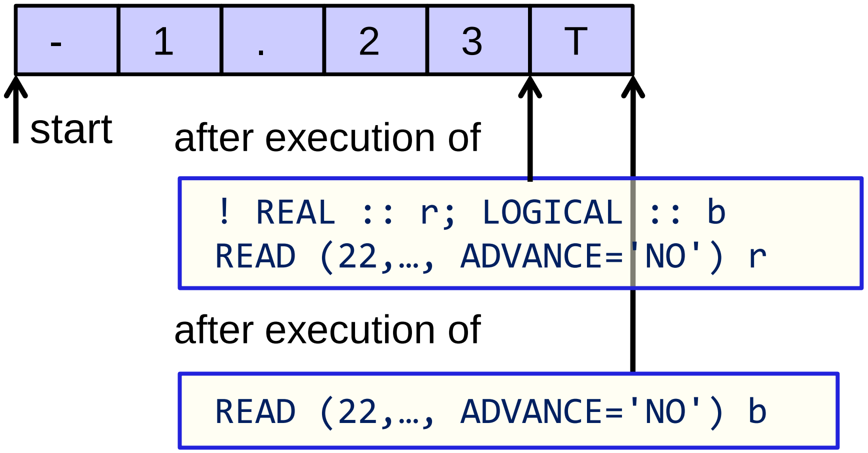
```
READ (…, ADVANCE='NO') …
```

(default setting is 'YES')

Let's use a magnifying glass

on record No. 2 ...

read with `'(f5.2)','(l1)'` – each square is 1 character (byte)



-	1	.	2	3	T

start

after execution of

```
! REAL :: r; LOGICAL :: b
READ (22,…, ADVANCE='NO') r
```

after execution of

```
READ (22,…, ADVANCE='NO') b
```

if a further READ statement is executed, it would abort with an end-of-record condition.

retrieve iostat-value (default integer) via iostat specifier: allows handling by user code and positions connection at beginning of next record:

```
READ (…,ADVANCE='NO',IOSTAT=ios) …
IF (is_iostat_eor(ios)) …
```

Nonadvancing I/O (2)

❑ Reading character variables

- the SIZE specifier allows to determine the number of characters actually read

```
CHARACTER(len=6) :: c
INTEGER :: sz
:
! Read chars from file into string:
READ(23,fmt='(a6)',advance='NO',&
    pad='YES', iostat=ios, size=sz) c
! Set remaining chars to
! a non-blank char if EOR occurs:
IF (is_iostat_eor(ios)) c(sz+1:)='X'
```

- mainly useful in conjunction with EOR (end-of-record) situations

❑ Nonadvancing writes

- usually used in form of a sequence of nonadvancing writes, followed by an advancing one to complete a record

❑ Final remarks

- nonadvancing I/O may not be used in conjunction with name-list, internal or list-directed I/O
- several records may be processed by a single I/O statement also in non-advanced mode
- format reversion takes precedence over non-advancing I/O

Issues with I/O for derived data types

- **Non-trivial derived data type**

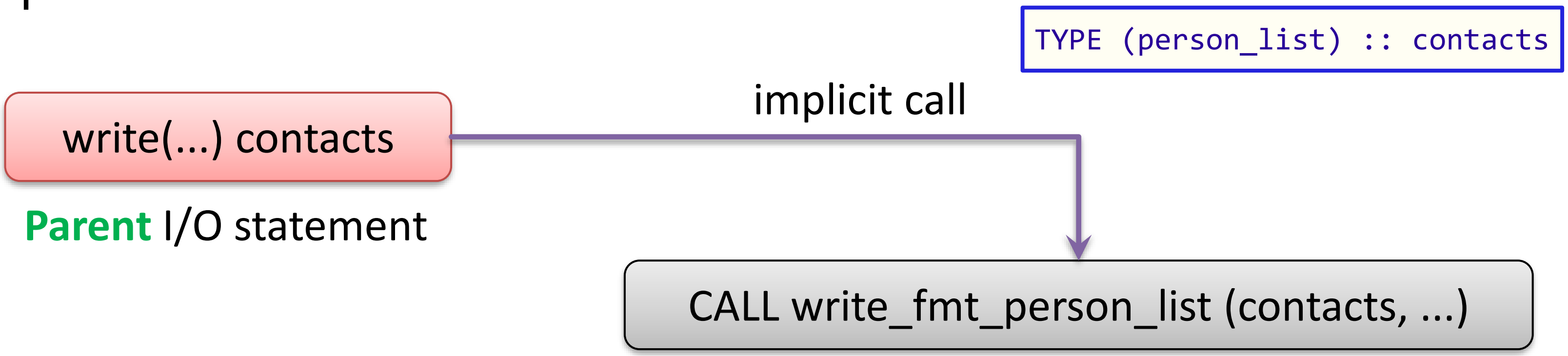
```fortran
MODULE mod_person
  TYPE :: person_list
    CHARACTER(len=:), ALLOCATABLE :: name
    INTEGER :: age
    TYPE(person_list), POINTER :: next
  END TYPE
...
```

- **An object of this type cannot appear directly in a data transfer statement**
- **Workaround:**
  - write module procedures that process type components individually
- **Disadvantages:**
  - F95 recursive I/O is disallowed (makes nesting of types difficult)
  - I/O transfer not easily integrable into an I/O stream
    - ➢ defined by edit descriptor for intrinsic types and arrays,
    - ➢ or a sequence of binary I/O statements

F03 User-defined Derived Type I/O

Concept:

- execution of a data transfer statement causes a **user-defined** procedure to be executed



```
TYPE (person_list) :: contacts
```

write(...) contacts

Parent I/O statement

implicit call

CALL write_fmt_person_list (contacts, ...)

- implementation:

```
RECURSIVE SUBROUTINE write_fmt_person_list(contacts, ...)
  ...
  WRITE(...) contacts%name
  ...
  IF (associated(contacts%next) &
      CALL write_fmt_person_list (contacts%next, ...)
```

Child I/O statement

Recursive invocation permitted

Binding I/O subroutines to derived types

- **Two variants are possible**

  1. Use an unnamed generic interface (required for non-extensible types)

```
INTERFACE write(formatted)
  MODULE PROCEDURE write_fmt_person_list
END INTERFACE
```

  2. Use a generic type-bound procedure

```
TYPE :: person_list
  :
CONTAINS
  GENERIC :: write(formatted) => write_fmt_person_list
END TYPE
```

- **Notes:**
  - more than one specific may exist (e.g. refer to kind parameters or type of object)
  - analogous: I/O binding declarations for `write(unformatted)`, `read(formatted)`, `read(unformatted)`

Client use

Formatted DTIO: the DT edit descriptor

```
TYPE(person_list) :: contacts
: ! set up contacts
: ! open formatted file to unit
WRITE(unit,FMT=,(DT “Person_List“ (4,20))', IOSTAT=is) contacts
: ! close unit and release resources for contacts
```

These two objects are transmitted to the user-defined routine as the **iotype** and **v_list** arguments, respectively

Unformatted DTIO

```
TYPE(person_list) :: friends
: ! unformatted writing also bound to person_list
: ! set up friends
: ! open unformatted (direct access) file to unit 21

WRITE(unit, REC=n) friends
```

DTIO restricted module procedure interface

procedure names are only placeholders

```
SUBROUTINE formatted_io(dtv,unit,iotype,v_list,iostat,iomsg)

SUBROUTINE unfmatted_io(dtv,unit,                iostat,iomsg)
```

- ❑ **`dtv`**
  - scalar of derived type
  - polymorphic iff type is extensible
  - of suitable **`intent`**
- ❑ **`unit`**
  - **`integer, intent(in)`** – describes I/O unit or is negative for internal I/O
- ❑ **`iotype`** (formatted only)
  - **`character, intent(in)`** **`'LISTDIRECTED', 'NAMELIST'`** or **`'DT'//string`** see **`dt`** edit descriptor
- ❑ **`v_list`** (formatted only)
  - **`integer, intent(in)`**- assumed shape array see **`dt`** edit descriptor
- ❑ **`iostat`**
  - **`integer, intent(out)`** – scalar, describes error condition
  - **`iostat_end / iostat_eor /`** zero if all OK
- ❑ **`iomsg`**
  - **`character(*)`** - explanation for failure if iostat nonzero

Limitations for DTIO subroutines

- **I/O transfers to other units than `unit` are disallowed**
  - I/O direction also fixed
  - Exception: internal I/O is OK (and commonly needed)
- **Inside a formatted DTIO procedure,**
  - I/O is **nonadvancing** (no matter what you specify for **`ADVANCE=`**)
- **Use of the statements**
  - **`OPEN, CLOSE, REWIND`**
  - **`BACKSPACE, ENDFILE`**

  is disallowed

- **File positioning:**
  - on entry: left tab limit
  - on return: no record termination
  - positioning with
    - **`REC=`**... (direct access) or
    - **`POS=`**... (stream access)

  is **disallowed**
  (it is implicitly determined by the Parent I/O statement)

Implementation details of DTIO routine

```
  : ! module mod_person continued
  RECURSIVE SUBROUTINE write_fmt_person_list (this,unit,iotype, &
                                              vlist,iostat,iomsg)
    CLASS(list_person), INTENT(IN)   :: this
    INTEGER,            INTENT(IN)   :: unit, vlist(:)
    CHARACTER(*),       INTENT(IN)   :: iotype
    INTEGER,            INTENT(OUT)  :: iostat
    CHARACTER(*),       INTENT(INOUT):: iomsg
    ! Local variable declarations not shown
    IF (iotype /= ’DTPerson_List’ .OR. size(vlist) < 2) THEN
      iostat = 42; iomsg=’Unsupported DT configuration’; RETURN
    END IF
    WRITE(pfmt, '(a,i0,a,i0)') '(i',vlist(1),',a',vlist(2),')'

    WRITE(unit, fmt=pfmt, iostat=iostat) this%age,this%name
    IF (iostat == 0 .AND. associated(this%next)) &
      CALL write_fmt_person_list (this%next,unit,iotype,&
                                  vlist,iostat,iomsg)
  END SUBROUTINE
    : ! other procedures
END MODULE mod_person
```

See **`examples/uddtio`**

Stream I/O

An access mode modeled on C streams:

```
OPEN (myunit, ..., ACCESS='STREAM', FORM='FORMATTED')
```

- usable for formatted and unformatted I/O
- for formatted stream I/O, there is no maximum record length. Explicit newlines can be written to terminate a record:

```
WRITE (myunit, FMT='(a)') str1, new_line('a'), str2
```

File positioning:

- on the granularity of a file storage unit
- explicit positioning may be supported:

```
INQUIRE (myunit, POS=current)
WRITE (myunit, FMT='(a)') str3
IF (...) WRITE(myunit, FMT='(a)'), POS=current, IOS=...) str4
```

INTEGER variable

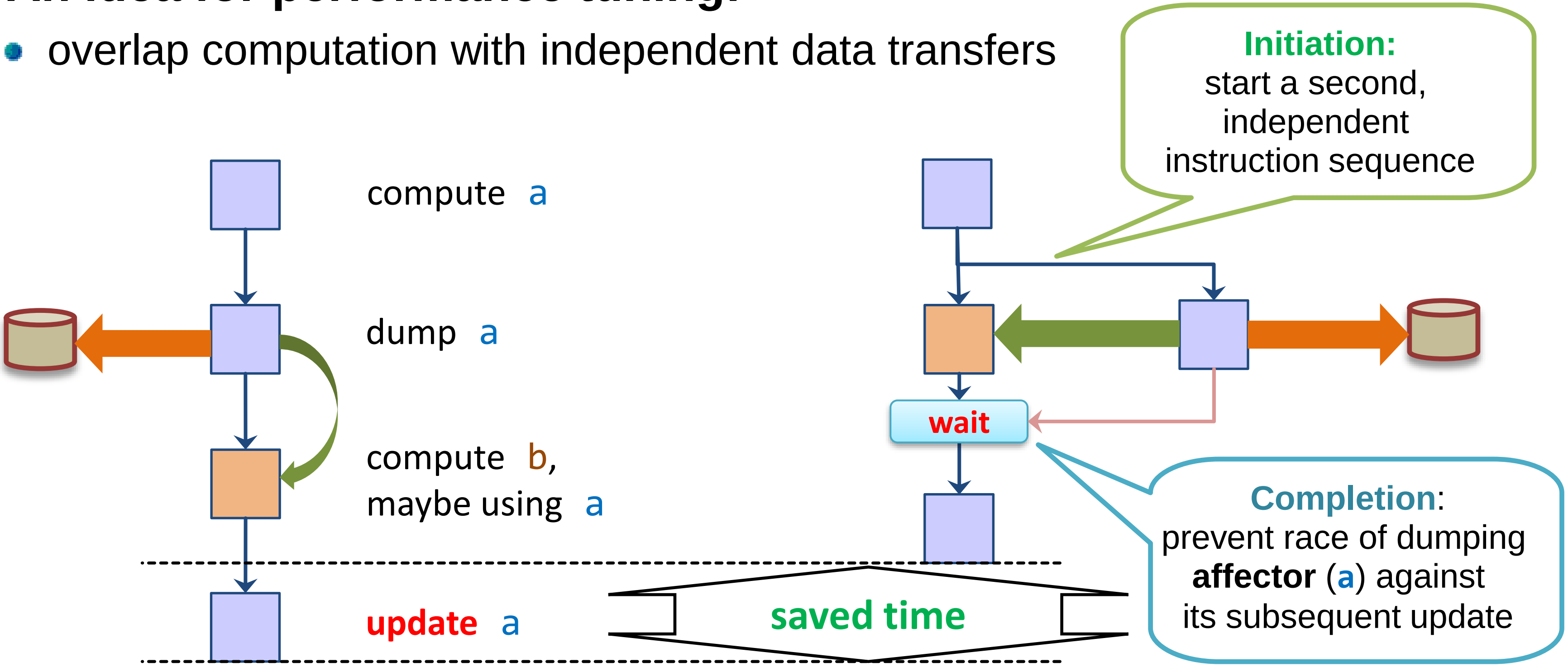
value from **INQUIRE** (or 1) must be used

- **str3** value is (maybe partially) overwritten; previous content is **preserved**

Asynchronous processing

An idea for performance tuning:

- overlap computation with independent data transfers



Initiation: start a second, independent instruction sequence

Completion: prevent race of dumping **affector** (a) against its subsequent update

Assumption:

- additional resources are available for processing the extra activity or even multiple activities (without incurring significant overhead)

The ASYNCHRONOUS attribute:
Contractual obligations between initiation and completion

Programmer:

- if affector is dumped, do not redefine it
- if affector is loaded, do not reference or define it
- analogous for changing the association state of a pointer, or the allocation state of an allocatable

Attribute syntax:

```fortran
REAL(rk), ASYNCHRONOUS :: x(:,:)
```

- here: for an assumed-shape array dummy argument
- sometimes also implicit (if the compiler can deduce it)

Processor:

- do not perform code motion of references and definitions of affector across initiation or completion procedure
- code motion across procedure calls between initiation and completion is prohibited, even if the affector is not involved in any of them

Constraints for dummy arguments

- assure that no copy-in/out can happen to affectors
- violations rejected by compiler, assuming the ASYNCHRONOUS attribute is properly specified

F03

Scenario 1: asynchronous I/O

Example: non-blocking READ

```
REAL, DIMENSION(ndim), ASYNCHRONOUS :: a
INTEGER :: tag
OPEN(NEW_UNIT=iu,...,ASYNCHRONOUS='yes')
...
READ(iu, ASYNCHRONOUS='yes', ID=tag) a
: !  do work on something else
WAIT(iu, ID=tag, IOSTAT=io_stat)
!  do work with a
   ...  = a(i)
```

no prefetches on a here

Actual asynchronous execution

- is at processors discretion
- likely most advantageous for large, unformatted transfers

Ordering requirements

- apply for a sequence of data transfer statements on the same I/O unit
- but not for data transfers to different units

`ID` specifier

- allows to assign each individual statement a tag for subsequent use
- if omitted, WAIT blocks until **all** outstanding I/O transfers have completed

INQUIRE

- permits non-blocking query of outstanding transfers via **`PENDING`** option

F18

Scenario 2: non-blocking MPI calls

Non-blocking receive - equivalent to a READ operation

```
REAL :: buf(100,100)
TYPE(MPI_Request) :: req
TYPE(MPI_Status) :: status
... ! Code that involves buf
BLOCK
  ASYNCHRONOUS :: buf
  CALL MPI_Irecv( buf, size(buf), MPI_REAL, src, tag, &
                  MPI_COMM_WORLD, req )
  ... ! Overlapped computation that does not involve buf
  CALL MPI_Wait( req, status )
  ... ! Code that involves buf
END BLOCK
```

asynchronous execution is limited to BLOCK

permitted, but may perform better outside the BLOCK

Likely a good idea to avoid call stacks with affector arguments

- violations of contract or missing attribute can cause quite subtle bugs that surface rarely

Performance considerations for using I/O

Expected types of I/O

1. **Configuration data**
   - usually small, formatted files
   - parameters and/or meta-data for large scale computations
2. **Scratch data**
   - very large files containing complete state information
   - required e.g., for checkpointing/restarting

   → rewrite in regular intervals
   - throw away after calculation complete
3. **Data for permanent storage**
   - result data set
   - for post-processing
   - to be kept (semi-) permanently
   - archive to tape if necessary
   - may be large, but do not (necessarily) comprise complete state information

Which file system(s) should I use?

For I/O of type 1:

- any will do
- if working on a shared (possible parallel) file system:

Beware transaction rates

→ OPEN and CLOSE stmts may take a long time

→ do not stripe files

For I/O of type 2 or 3:

- need a high bandwidth file system

→ parallel file system with block striping

- large file support nowadays is standard

What bandwidths are available?

- normal SCSI disks

~100 MByte/s

- DSS storage arrays at LRZ:

up to 7 GByte/s

- SuperMUC storage arrays:

up to 300 GByte/s

  - aggregate for all nodes
  - single node can do up to 2 GB/s (large files striped across disks)

→ writing the memory content of system to disk takes ~40 minutes

I/O formatting issues
various ways of reading and writing

better performance

Formatted I/O

- list directed

```
write(unit,fmt=*) ...
```

- with format string

```
write(unit,fmt=`(es20.13)`) ...
write(unit,fmt=iof) ...
```

→ can be static or dynamic

Unformatted I/O

- sequential

```
write(unit) ...
```

- direct access

```
write(unit, rec=i) ...
```

→ can also be formatted

I/O access patterns

better performance

- by implicit loop `write(...) ((a(i,j),i=1,m),j=1,n)`
- by array section `write(...) a(1:m,1:n)`
- by complete array `write(...) a`

I/O performance for implicit DO loops

- **Improve performance by**
  - imposing correct loop order (fast loop inside!)
  - more important: writing large block sizes

```
do i=1,16
  write(unit[,...]) (a(i,k),k=1,10000000)
end do
```

Large blocks, but wrong order.
On some platforms this may give a performance hit
→ re-copy array or reorganize data

  - proper tuning

    → performance may exceed that for array sections

Discussion of unformatted I/O properties

- **No conversion needed**
  - saves CPU time
- **No information loss**
- **Needs less space on disk**
- **File not human-readable**
  - binary
  - Fortran record control words
    - possible interoperability problems with I/O in C
    - convert to Stream I/O
- **Format not standardized**
  - in practice much the same format is used anyway
  - exception big/little endian issues
  - solvable if all data types have same size

- **Support for little/big endian conversion by Intel compiler**
  - enable at run time
  - suitable setting of environment variable F_UFMTENDIAN
  - example:

```
export F_UFMTENDIAN=“little;big:22”
```

will set unit 22 **only** to big-endian mode

(little endian is default)

- performance impact??
- other compilers might need:
  - changes to source or
  - compile time switch

Buffering setup for Intel compilers

- **Setting up buffering as follows can significantly increase I/O performance:**

  ```
  export FORT_BUFFERED=true
  ```

  - this will activate buffering for all I/O units
- **Blocksize**
  - this is a tunable. On LRZ HPC systems, we recommend
  - Linux Cluster SCRATCH:

    ```
    export FORT_BLOCKSIZE=8388608
    ```

  - SuperMUC-NG SCRATCH/WORK:

    ```
    export FORT_BLOCKSIZE=16777216
    ```

I/O and program design

- **Except for debugging or informational printout**
  - try to encapsulate I/O as far as possible

    → each module has (as far as necessary) I/O routines related to it’s global data structures

    → mapping of file names should reflect this
  - write extensibly, i.e.: use a generic interface which can then be applied to an extended type definition
    - in fact module internal code can usually be re-used
    - keep in mind: performance issues may crop up if code used outside its original design point
- **Additional documentation requirement**
  - description of structure of data sets needed

following now: Exercise session 3

Parameterized derived Types

Parameterized Derived Types: Introduction

- ❑ So far we have seen three important concepts related to OOP-paradigm: inheritance, polymorphism and data encapsulation
- ❑ Here we add another concept:
  - ➢ Concept of a **parameterized derived type**
- ❑ We know the concept already, have a look at object declarations of intrinsic type:

```fortran
! scalar of type real
! with non-default kind:
real(kind=real32) :: a
! array of integer numbers
! with non-default kind parameter
integer(kind=int64) :: numbers(n)
!character of default kind
!with deferred length parameter:
character(len=:), allocatable :: path
```

- ● All intrinsic types are actually parameterized with the kind parameter (intrinsic types: integer, real, complex, logical, character)
- ● Objects of type character are additionally parameterized with the len parameter
- ● We extend the concept to derived types, e.g.:

F03

```fortran
!define parameterized type:
type pmatrixT(k,r,c)
   integer, kind :: k
   integer, len :: r,c
   real(kind=k) :: m(r,c)
end type
!declare an object of that type
type(pmatrixT(real64,30,20)) :: B
```

Parameterized Derived Types:
Kind and Length Parameters

- F2003 permits type parameters of derived type objects.
  Two varieties of type parameters exist:
  - kind parameters, must be known at compile time
  - Length parameter which are also allowed to be known only during runtime

```fortran
!kind parameters from intrinsic module
use iso_fortran_env, only: real32, real64
!define parameterized type:
type pmatrixT(k,r,c)
   integer, kind :: k
   integer, len :: r,c
   real(k) :: m(r,c)
end type
!: declare an object of that type
type(pmatrixT(real32,30,20)) :: A
type(pmatrixT(real64,10,15)) :: B
```

- Type parameters are declared the same way as usual DT-components with the addition of specifying either the kind or len attribute
  - k here resolves to compile-time constant real32 (for A) and real64 (for B)
  - r,c could be deferred but here resolves to literal constants 30,20 (A) and 10,15 (B)

Parameterized Derived Types:

Parameterized Derived Type vs. Conventional Derived Type

```
module mod_pmatrix
!define parameterized type:
  type pmatrixT(k,r,c)
    integer, kind :: k
    integer, len :: r,c
    real(k) :: m(r,c)
  end type
contains
subroutine workona_pmat32(cs,rs)
    integer :: cs,rs
    type(pmatrixT(real32,cs,rs)) :: M
    !M%m(:,:) = …
  end subroutine
subroutine workona_pmat64(cs,rs)
    integer :: cs,rs
    type(pmatrixT(real64,cs,rs)) :: M
    !M%m(:,:) = …
  end subroutine end module
end module
```

advantage:
1 single type definition
2 dynamic data in component without allocatable or pointer attribute

```
! client use
call workona_pmat32(20,30)
call workona_pmat64(20,30)
```

```
module mod_matrix
  type matrix32T
    real(real32),allocatable:: m(:,:)
  end type
 type matrix64T
    real(real64),allocatable:: m(:,:)
  end type
contains
  subroutine workona_mat32(cs, rs)
    type(matrix32T) :: M
    allocate(M%m(cs,rs))
    !M%m(:,:) = …
  end subroutine
 subroutine workona_mat64(cs, rs)
    type(matrix64T) :: M
    allocate(M%m(cs,rs))
    !M%m(:,:) = …
  end subroutine
end module
```

disadvantage:

1 two type definitions
2 dynamic data only through allocatable or pointer attribute

```
! client use
call workona_mat32(20,30)
call workona_mat64(20,30)
```

Parameterized Derived Types:
Inquire Type parameters

- Type parameters of a parameterized object can be accessed directly using the component selector

```
!type definition as in previous example
type(pmatrixT(real64,cols,rows)) :: A

write(*,*) A%k
write(*,*) A%c
write(*,*) A%r
do i = 1,A%c
  do j = 1,A%r
    A%m(i,j) = …
  enddo
enddo
```

- However, type parameters cannot be directly modified, e.g.:

```
type(pmatrixT(real64,cols,rows)) :: A
A%k=real32 ! invalid
A%c=8      ! invalid
A%r=12     ! invalid
```

Parameterized Derived Types:
Assumed Type Parameters

- ❑ Let‘s pass a parameterized object into a subroutine

```
!type definition as in previous example
type(pmatrixT(real64,20,30)) :: A
type(pmatrixT(real64,10,20)) :: B

call proc_pmat(A)
call proc_pmat(B)
```

- ❑ The len parameter can be assumed from the actual argument using the *-notation
- ❑ NOTE! The kind parameter cannot be assumed!
  - ➢ But dealing with the (few) different kind parameters of interest is potentially more manageable than having to additionally deal with all len-parameter combinations
- ❑ NOTE! Type parameters cannot be assumed if dummy object has the allocatable or pointer attribute

```
module mod_pmatrix
 !: definitions as before
 interface proc_pmat
   module procedure :: proc_pmat32, &
            proc_pmat64
 end interface
contains
  subroutine proc_pmat64(M)
    ! dummy with assumed len parameters:
    type(pmatrixT(real64,*,*)) :: M
    do i = 1,M%c
       do j = 1,M%r
          M%m(i,j) = …
       enddo
     enddo
  end subroutine
 subroutine proc_pmat32(M)
   type(pmatrixT(real32,*,*)) :: M
 end subroutine
 !:
 subroutine otherwork_pmat64(M1,M2)
   type(pmatrixT(real64,*,*)), &
                 allocatable :: M1 !invalid
   type(pmatrixT(real64,*,*)), &
                     pointer :: M2 !invalid
 end subroutine
end module
```

Parameterized Derived Types: Deferred Type Parameters

- Using the colon notation we may declare objects of parmeterized derived type with deferred len-parameter if they have the pointer or allocatable attribute

```
!type definition as in previous example
type(pmatrixT(real32,:,:)), allocatable :: A, B
type(pmatrixT(real32,:,:)), pointer :: P
type(pmatrixT(real32,5,8)) :: M_5_8

allocate(type(pmatrixT(real32,15,10)::A)
P => M_5_8
allocate(B, source=P) !B allocated B%r=5, B%c=8
```

- The previous invalid code (assumed len parameter for allocatable dummy object) can be corrected using deferred len parameters using colon-notation for passed dummy objects with allocatable or pointer attribute

```
module mod_pmatrix
 !: definitions as before
contains
!:
 subroutine otherwork_pmat64(M1,M2)
   type(pmatrixT(real64,:,:)), allocatable :: M1 ! valid
   type(pmatrixT(real64,:,:)), pointer     :: M2 ! valid
 end subroutine
!:
end module
```

Parameterized Derived Types:
Default Type Parameters

- It is possible to define default parameters for a parameterized derived type

```fortran
type pmatrixT(k,r,c)
   integer, kind :: k=real64
   integer, len :: r=6,c=6
   real(k) :: m(r,c)
end type
! you may declare objects of such a type
! without specifying all parameter values, e.g.:
type(pmatrixT)          :: default_matrix ! all parametes default
type(pmatrixT(real32))  :: real32_matrix  ! with default len, specific kind
type(pmatrixT(r=3,c=9)) :: matrix_3_9     ! with default kind, specific len
type(pmatrixT(c=9,r=3,k=real32)) :: out_of_order ! Out of order specification
                                                 ! using keywords
```

- You may specify only a subset of parameters and/or out of order but it requires to use keyword notation to correctly associate each actual parameter with the right type-parameter

- This also applies to deferred or assumed len declarations:

```fortran
type(pmatrixT(k=real32,c=*,r=*)) :: M_assumed
type(pmatrixT(c=:,r=:,k=real32)), allocatable :: M_deferred
type(pmatrixT(c=:,r=:,k=real32)), pointer     :: M_pointer
```

Parameterized Derived Types:
Inheritance and polymorphism

- ❑ It is possible to inherit properties from an existing base type via type extension
- ❑ Extended types may add additional kind and/or len parameters for subsequent component declarations

```fortran
type mat_aT(k,r,c)
   integer, kind :: k=real64
   integer, len :: r=1,c=1
end type
type,extends(mat_aT) :: mat_rT
   real(k) :: m(r,c)
end type
type,extends(mat_aT) :: mat_crT(k2,ml)
   real(k) :: m(r,c)
   integer, kind :: k2=int64
   integer, len :: ml=100
   integer(k2) :: counter(r,c)
   character(len=ml) :: message
end type
```

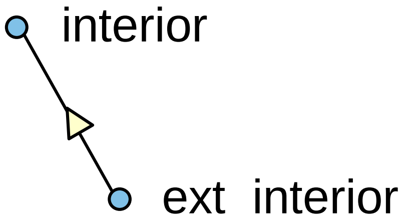
```fortran
! usage, e.g.:
type(mat_rT(real32,9,9)), target :: A
type(mat_crT(real64,:,:,int32,:),  &
                  allocatable, target :: B
Class(*), pointer :: P

P => A ! P is now of dynamic type mat_rT
allocate(mat_crT(real64,5,5,int32,80) :: B)
P => B ! P is now of dynamic type mat_crT

! unwrap polymorphism to access components
select type(P)
type is (mat_crT(real64,*,*,int32,*))
  write(*,*)'%m=',P%m
  write(*,*)'%counter=',P%counter
end select
```

- ❑ unwrap polymorphism from polymorphic object (here P) to access components
- ❑ argument for type-guard statement: need to specify all kind parameters (compile-time constants) and all len parameters as assumed (*-notation)

Creation and Destruction of objects

Polymorphic type components

- **Assuming the following:**



- **and the type definitions**

```
TYPE :: t_poly
  CLASS(interior), &
         ALLOCATABLE :: r(:)
end type
```

```
TYPE :: t_unlimited
  CLASS(*), &
         ALLOCATABLE :: r(:)
END TYPE
```

- **Then the following constructors can be used:**

```
TYPE(t_poly) :: o_1
TYPE(t_unlimited) :: o_2

o_1 = t_poly([ &
  (interior(real(i)),i=1,3) ])
```

`o_1%r` is of dynamic type `interior`

```
o_1 = t_poly([ &
  (ext_interior(real(i)),i=1,3) ])
```

`o_1%r` is of dynamic type `ext_interior`

```
o_2 = t_unlimited([ &
  (interior(real(i)),i=1,3) ])
```

`o_2%r` is of dynamic type `interior`

- auto-(re)allocation occurs at each assignment
- difference `o_1` vs `o_2`: `o_2%r` can be of any type

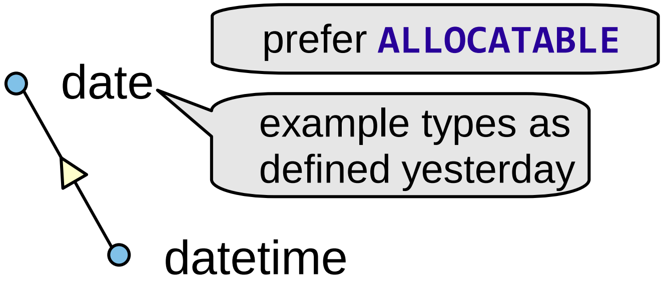
Polymorphic overloaded constructor (1)

- **Assumption:**
  - type of object to be created is not known at compile time

  possible reason: object's type is determined from information stored in an external file

  - how should the constructor be written in this case?

- **Need a polymorphic function result**
  - this must have the **`POINTER`** or **`ALLOCATABLE`** attribute



prefer **`ALLOCATABLE`**

date

example types as defined yesterday

datetime

Polymorphic overloaded constructor (2)

- **Specific function for base type `date()` overload:**

```
FUNCTION dt_io(fname) RESULT(this)
  CLASS(date), ALLOCATABLE :: this
  CHARACTER(LEN=*), INTENT(IN) :: fname
  CHARACTER(LEN=strmx) :: this_type
  : ! open file fname on unit
  READ(unit, …) this_type
  SELECT CASE (trim(this_type))
  CASE ('date')
    ALLOCATE(date :: this)
  CASE ('datetime')
    ALLOCATE(datetime :: this)
  CASE DEFAULT
    STOP 'unknown type'
  END SELECT
  : ! continued to the right
```

```
SELECT TYPE(this)
  TYPE IS (date)
    : ! read and set up date
  TYPE IS (datetime)
    : ! read and
    : ! set up datetime
  END SELECT
  : ! close file
END FUNCTION dt_io
```

Usage of the polymorphic overloaded constructor

Target object is polymorphic

```
USE mod_date
CLASS(date), ALLOCATABLE :: o_d

ALLOCATE(o_d, source=date('D.dat'))
```

- assignment to polymorphic variable is not allowed in F03
- in F08, the last line of the above code can be replaced by

```
o_d = date('D.dat')
```

(auto-allocation of LHS to the type of the RHS; furthermore the RHS may also involve the object appearing on the LHS – this is not allowed in sourced allocation)

Target object is non-polymorphic

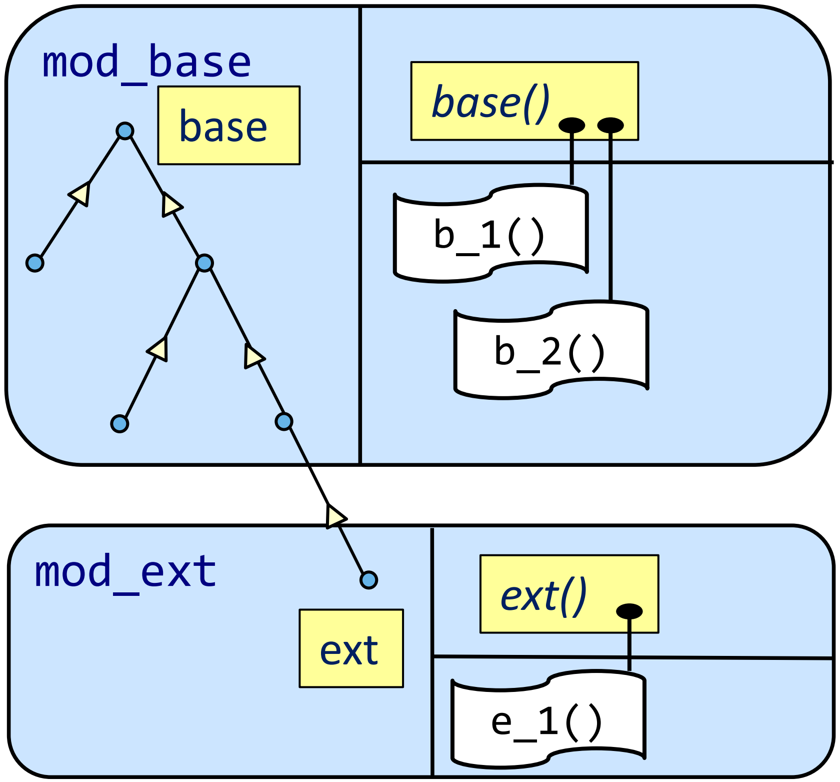
```
USE mod_date
TYPE(date) :: o_nonpoly

o_nonpoly = date('D.dat')
```

- type of LHS must be base type
- if the constructor produces an extension, the object will be truncated to the base type object

Covering the Inheritance Tree

- may require use of a submodule if extensions are defined in a different module
- alternatively, overload an extension via its name

Diagramming the polymorphic constructor



- **Type** base **could also be abstract**

- **Usage looks as follows** (whether or not base is abstract)**:**

```
USE mod_base
CLASS(base), ALLOCATABLE :: o
ALLOCATE(o, source=base(…))
SELECT TYPE (o)
TYPE IS (…)
: ! process object
TYPE IS (…)
: ! process object
CLASS DEFAULT
: ! throw error
END SELECT
```

decide on whether b_1 or b_2 is invoked

each clause must reference an **extension** of **base** if the latter is abstract

- **Illustrates going beyond module boundaries with an extension**

Factory methods with polymorphic dummy arguments

- **use the POINTER or ALLOCATABLE attribute**
  - third variant of polymorphism

```
SUBROUTINE produce(o_up, o_dt, …)
  CLASS(*), ALLOCATABLE :: o_up
  CLASS(date), ALLOCATABLE :: o_dt
  : ! determine type for o_up
  IF (…) THEN
    ALLOCATE(body :: o_up)
  ELSE IF (…)
  :
  END IF
  : ! determine type for o_dt
  IF (…) THEN
    ALLOCATE(datetime :: o_dt)
  ELSE IF (…)
  :
  END IF
end subroutine
```

prefer **ALLOCATABLE**
→ avoid memory leaks

- **Invocation of the procedure**

```
CLASS(*), ALLOCATABLE :: o1
CLASS(date), ALLOCATABLE :: o2

CALL produce(o1, o2, …)
```

- **Actual argument**
  - must have **same** attribute,
  - and **same** declared type

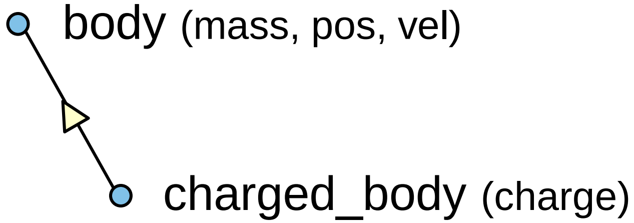
as the dummy argument

(otherwise, type compatibility could be violated)

- **Note:**
  - such a procedure cannot be bound to a type via one of the allocatable arguments (→ see day 2)

Returning to overloaded operators: handling polymorphic result variables

- **Example:**



  - form the sum of two bodies

$$m = m_1 + m_2$$

$$\vec{r} = (m_1\vec{r_1} + m_2\vec{r_2})/(m_1 + m_2)$$

etc.

```
TYPE :: body
  : ! data components
CONTAINS
  PROCEDURE :: plus
  GENERIC :: OPERATOR(+) => plus
END TYPE
TYPE, EXTENDS(body) :: charged_body
  REAL :: charge
CONTAINS
  PROCEDURE :: plus => plus_charged
END TYPE
```

override for extension

- **Implementation of TBP:**

```
FUNCTION plus(b1, b2)
  CLASS(body), INTENT(IN) :: b1, b2
  CLASS(body), ALLOCATABLE :: plus

  ALLOCATE(body :: plus)
  plus%mass = b1%mass + b2%mass
  plus%pos = …
  plus%vel = …
END FUNCTION plus
```

  - overriding this TBP is required for **each** extension of body

Overriding a specific in the polymorphic generic
(for a symmetric implementation)

```
FUNCTION plus_charged(b1, b2)
  CLASS(charged_body), &
         INTENT(IN) :: b1

  CLASS(body), INTENT(IN) :: b2
  CLASS(body), ALLOCATABLE :: &
         plus_charged
```

declarations for second argument and function result are forced by restrictions on TBP interface

Nested `select type` statement are needed in order to access the type components

continued from left panel ...

```
  SELECT TYPE(b2)
  CLASS IS (charged_body)
    ALLOCATE(charged_body :: &
             plus_charged)
    SELECT TYPE (plus_charged)
    TYPE IS (charged_body)
      plus_charged%…  = …
    END SELECT
  CLASS DEFAULT
    STOP 'Parent of charged_body. &
          & Aborting.'
  END SELECT
END FUNCTION plus_charged
```

Usage of the operator and its overridden version

```
CLASS(body), ALLOCATABLE :: o1, o2, b1, b2

ALLOCATE(body :: b1, b2)
: ! give values to b1, b2

o1 = b1 + b2
```

invokes `plus((b1),(b2))`

```
DEALLOCATE(b1, b2)
ALLOCATE(charged_body :: b1, b2)
: ! give values to b1, b2

o2 = b1 + b2
```

invokes `plus_charged((b1),(b2))`

- unless F08 support for polymorphic LHS is implemented, the above also requires overloading of the assignment operator (exercise)

Remember Finalizers

- **Have a class or object associated with additional state**
  - open files
  - unfinished non-blocking network (MPI) calls
  - allocated pointer components
- **Imagine object goes out of scope**
  - unrecoverable I/O unit
  - communication breakdown
  - memory leak
- **Solution: object auto-destructs by**
  providing it with a procedure which is called as object
  - goes out of scope,
  - is deallocated,
  - is passed to an `INTENT( out )` dummy argument, or
  - appears on the left hand side of an intrinsic assignment

Syntax of Final Procedure

- **Type definition**

```
TYPE :: sparse
  :
  TYPE(sparse), POINTER :: &
        next => null()
CONTAINS
  FINAL :: finalize_sparse
END TYPE
```

- **Finalizer implementation:**

applicability to array objects

```
ELEMENTAL RECURSIVE subroutine &
          finalize_sparse(this)
  TYPE(sparse), INTENT(INOUT) :: &
          this
  IF (associated(this%next)) THEN
    DEALLOCATE(this%next)
  END IF
END SUBROUTINE
```

assumes that all targets have been dynamically allocated

Differences to TPB:

- **Not normally invoked by programmer**
  - finalizer is automatically executed as described on previous slide
- **Must have single dummy argument**
  - of type to be finalized
  - non-polymorphic
  - non-pointer, non-allocatable
  - all length type parameters assumed
- **Generic set of finalizers possible:**
  - rank
  - kind parameter values
  - multiple execution order processor-dependent

The IMPURE attribute and one of its applications

F08

- **By default, ELEMENTAL procedures must not have side effects**
  - consequence: in many cases, no elemental finalizer can be written
  - specifying the **IMPURE** attribute allows to circumvent this restriction

- **Example:**

type introduced earlier

```
TYPE, EXTENDS(handle) :: file_handle
  PRIVATE
  INTEGER :: unit
  CLASS(h), POINTER :: data
CONTAINS
  PROCEDURE :: open => file_open
  FINAL :: delete_fh
END TYPE file_handle
```

- **Finalizer with side effects:**

```
IMPURE ELEMENTAL subroutine &
       delete_fh(this)
  TYPE(file_handle) :: this
  IF (this%state == 1) THEN
    IF associated(this%data)) &
      WRITE(this%unit, …) this%data
    CLOSE(this%unit)
    this%state = 0
  END IF
END SUBROUTINE
```

e.g., use UDDTIO

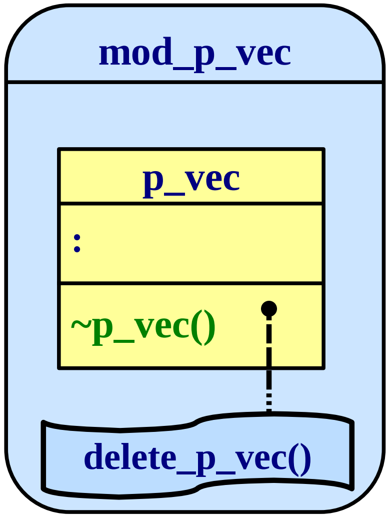
- **Usage:**

```
SUBROUTINE foo(...)
  TYPE(file_handle) :: fh0, fh1(5)
  ...


END SUBROUTINE
```

initialize **local** variables fh0, fh1 and associate with data

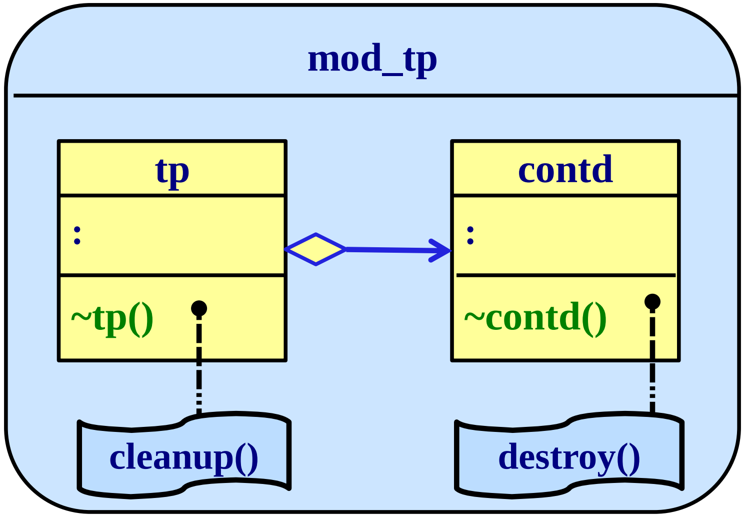
invokes **delete_fh** for **both** fh0 and fh1

Diagrammatic representation of finalizers



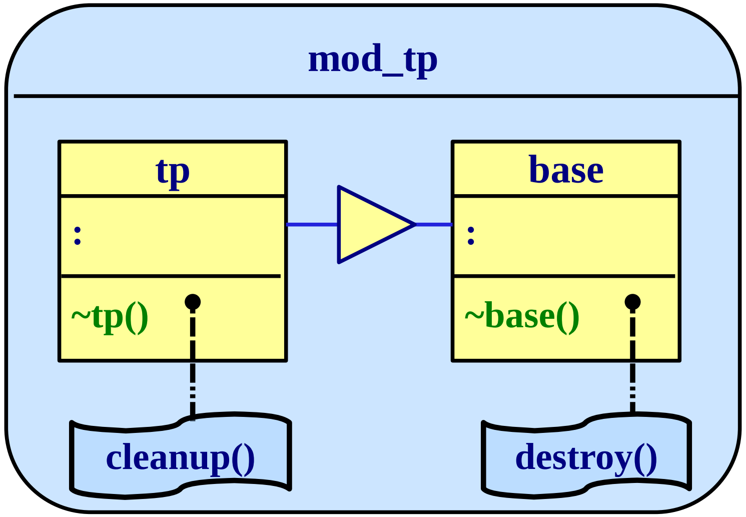
- **Finalizer is not inherited by extensions**
  - reflected in nonpolymorphic argument
  - exact type match required

- **Layering of finalizers**



- **If an object of type `tp` goes out of scope**
  - first `cleanup()` is called
  - then `destroy()`
  - if `contd` is a pointer component, it needs to be explicitly deallocated or nullified in `cleanup()`

Finalization of type extensions



- **If `tp` is a subclass of base, and an object of type `tp` goes out of scope**
  - first `cleanup()` is called
  - then `destroy()`
- **This applies recursively in the case of more than one inheritance level**

Constraints on PURE procedures

Partly a F08 corrigendum

- **Procedure body:**
  - must not contain statements that cause an impure finalizer to be invoked
- **INTENT(OUT) argument:**
  - must not be polymorphic
- **PURE function:**
  - function result must not be an allocatable polymorphic entity
- **Quite heavy restrictions – reason:**
  - cannot check for possible invocation of impure finalizer at **compile time,** which is required for PURE

Returning to the Environment problem

Case study

Handling numerical integration

or

Using Polymorphism in the context of function arguments

Example: Numerical integration (1)
(cf „Modern Fortran Explained“, Section 14.9)

- **Quadrature routine**
  - usually provided with user defined function as dummy argument
- **Not flexible enough**
  - user-defined function interfaces typically do not fit required profile
  - want additional parameters
- **Available solutions**
  - use module globals (threading?)
  - additional dummy in quadrature routine
  - still not flexible
  - reverse communication interface
  - avoids function parameter
  - return from quadrature routine to request function data
  - complicated to use and implement
- **Object oriented solution**
  - define an interface class
  - encapsulate additional user data into type extension

Numerical integration example (2)

Defining the interface class

```
MODULE qdr
  TYPE, ABSTRACT :: qdr_fun
!   user-defined data elements in extension
  CONTAINS
    PROCEDURE(qdr_if), DEFERRED :: eval
  END TYPE
  ABSTRACT INTERFACE
    REAL(kind=rk) FUNCTION qdr_if(this, x)
      IMPORT :: qdr_fun
      CLASS(QDR_FUN) :: this
      REAL(kind=rk), INTENT(IN) :: x
    END FUNCTION
  END INTERFACE
  : ! further type definitions
CONTAINS
: ! continued
```

- **Programmer of client must implement `eval()` in own extension**
  - interface for this is also fixed
  - reason: is used in contained module procedure

Numerical integration example (3)
... and its module procedure

```fortran
REAL(KIND=rk) FUNCTION integral_1d(intv,fun,status)
  REAL(KIND=rk), INTENT(IN) :: intv(2)
  CLASS(qdr_fun), INTENT(IN) :: fun
  INTEGER, OPTIONAL, INTENT(OUT) :: status
  :
  DO …
    x = …
    y = fun%eval(x)
    :
  END DO
  :
  integral_1d = …
END FUNCTION
```

start default integration algorithm

- **Implementation does not (and should not) reference additional user data**
  - these are handed through to the overriding TBP via the **`fun`** object

Numerical integration example (4)

Subtyping in user code

- **Suppose function is a polynomial:** $\sum_{i=1}^{n} f_i \cdot x^{i-1}$

```
MODULE qdr_poly
  USE qdr
  TYPE, EXTENDS(qdr_fun) :: poly_fun
    REAL(KIND=rk), ALLOCATABLE :: f(:)
  CONTAINS
    PROCEDURE :: eval => eval_poly
  END TYPE
CONTAINS
  REAL(KIND=rk) FUNCTION eval_poly(this, x)
    CLASS(poly_fun) :: this
    REAL(KIND=rk), INTENT(IN) :: x
    : ! use Horner's scheme to evaluate
  END FUNCTION
END MODULE
```

Numerical integration example (5)

Usage by program

```
PROGRAM myprog
  USE qdr_poly
  TYPE(poly_fun) :: o_poly_fun
  REAL(KIND=rk) :: result
  :
  o_poly_fun%f = [ 1.0_rk, 2.5_rk, 4.0_rk ]
  result = integral_1d( [ -1._rk, 1._rk ], o_poly_fun )
  :
END PROGRAM
```

- **Can now extend to various methods for interpolation**
  - polynomial
  - spline
  - trigonometric

  or use other (arbitrary or analytical) representation

Numerical integration example (6)
Extending the functionality

Consider special cases

- integrals could be calculated analytically / faster
- discontinuous or singular integrands
- would like to be able to use alternative integrator

  (included with module)
- **example:** integral equation $\mu \int_0^1 K(x,t) f(t) dt + g(x) = f(t)$

  extend interface class slightly e.g.

```
  TYPE, ABSTRACT :: qdr_fun
    CLASS(qdr_opt), POINTER :: options => null()
!   module-defined subtypes determine dispatch
!   user-defined data elements in extension
  CONTAINS
    PROCEDURE(qdr_if), DEFERRED :: eval
  END TYPE
```

- previous functionality **unchanged**

Numerical integration example (7)
configuring options

- **Use an abstract type**
  - extend it for the specific purpose

```
TYPE, ABSTRACT :: qdr_opt
END TYPE

TYPE, EXTENDS(qdr_opt) :: &
          qdr_opt_sing
  REAL, ALLOCATABLE :: rs(:)
END TYPE
```

- **Reason:**
  - additional information is needed by the specific integrator (e.g. location of singularities)

- **Interface for specific integrator**

```
SUBROUTINE integral_1d_sing( &
         intv,fun,sing,status)
  REAL(rk), INTENT(IN) :: intv(2)
  CLASS(qdr_fun) :: fun
  TYPE(qdr_opt_sing) :: sing
  INTEGER, INTENT(OUT) :: status
END SUBROUTINE
```

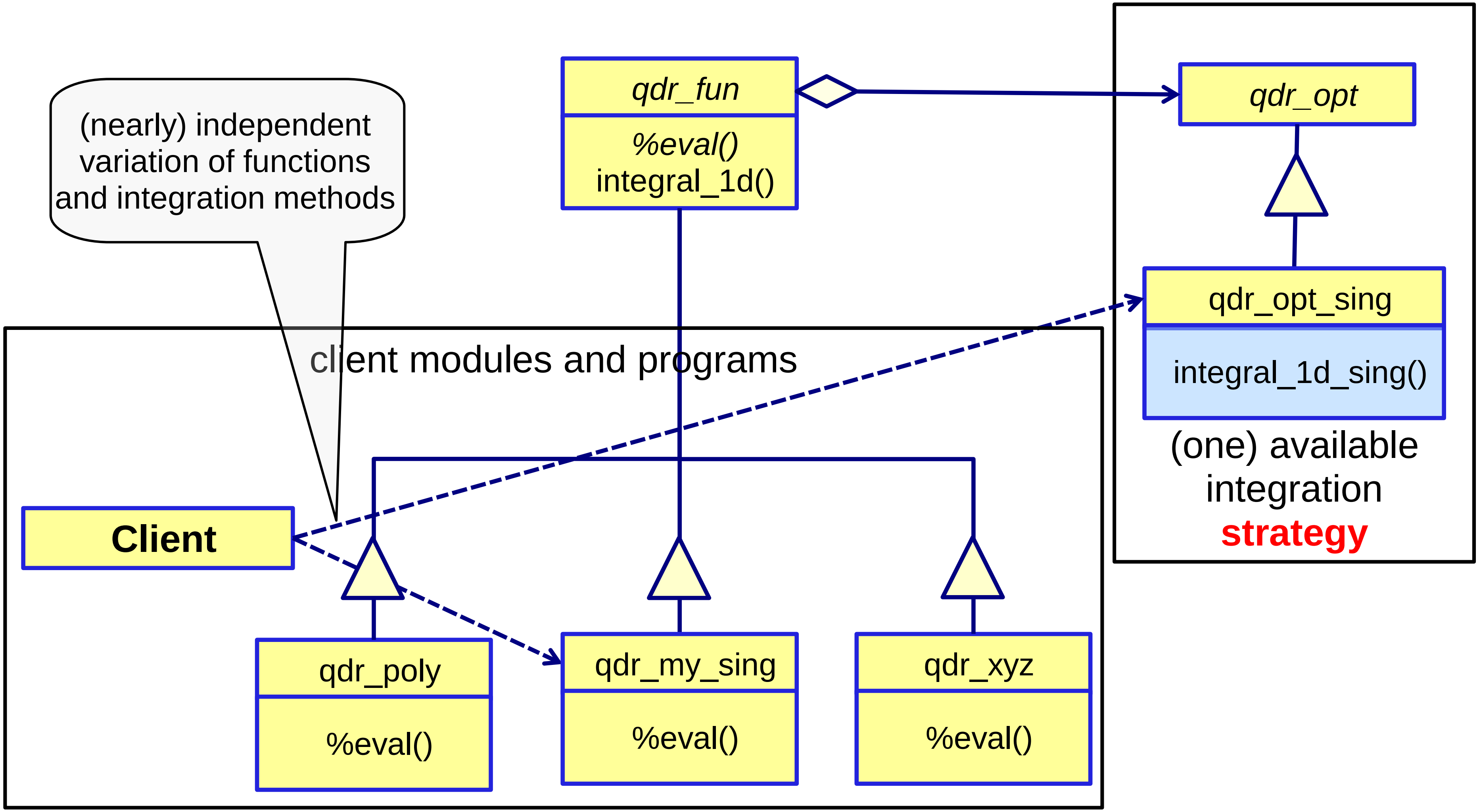
Numerical integration example (8)

Updates on integral_1d

```fortran
REAL(KINd=rk) FUNCTION integral_1d( intv, fun, status )
  REAL(KIND=rk), INTENT(IN) :: intv(2)
  class(qdr_fun), intent(in) :: fun
  INTEGER, OPTIONAL, INTENT(OUT) :: status

  IF (associated(fun%options)) THEN
     SELECT TYPE (fun%options)
     TYPE IS (qdr_opt_sing)
       call integral_1d_sing(intv,fun,fun%options,status)
     : ! continue dispatch
     END SELECT
  ELSE
    : ! start default algorithm
    DO ...
      y = fun%eval(x)
      :
    END DO
    :
    integral_1d = ...
  END IF
END FUNCTION
```

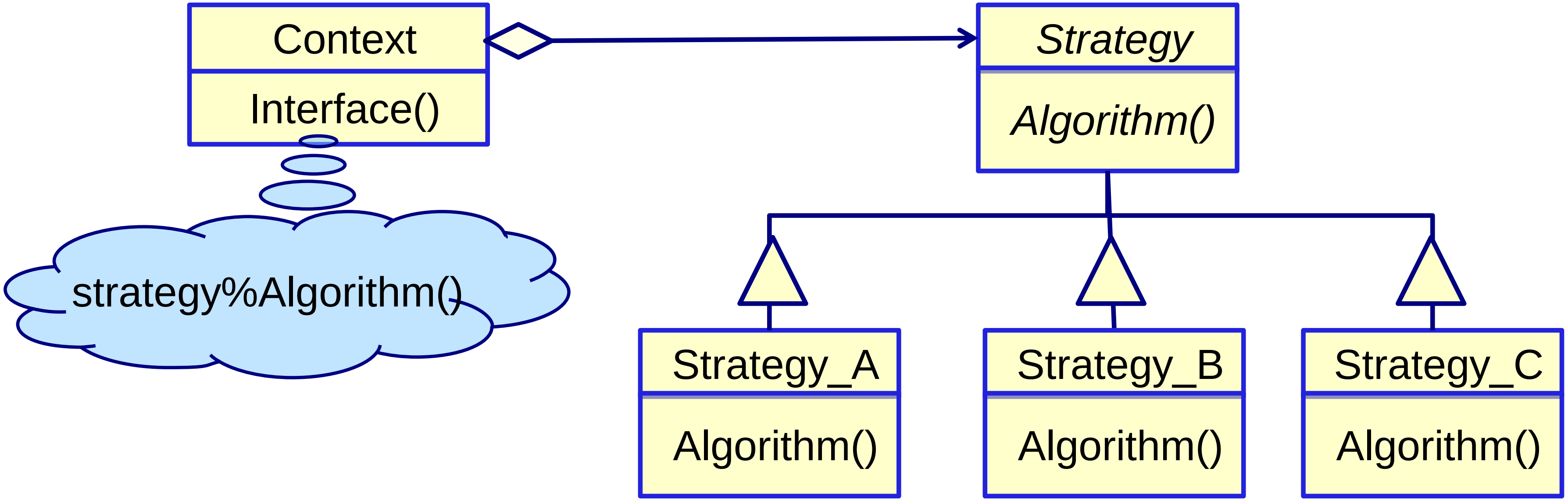
other specialized integrators
(and don't forget CLASS DEFAULT)

Diagramming the integration interface


(nearly) independent variation of functions and integration methods
qdr_fun
%eval()
integral_1d()
qdr_opt
qdr_opt_sing
integral_1d_sing()
(one) available integration **strategy**
client modules and programs
Client
qdr_poly
%eval()
qdr_my_sing
%eval()
qdr_xyz
%eval()

Strategy Pattern: Varying algorithms transparently

- **Replaces subclassing for variation**
  - context interface provides appropriate support
  - references only to abstract strategy (dependency inversion)
- **Fewer classes, but more objects in application**
- **A “behavioral” pattern**



Adapting legacy code (1)

- **Assumption: A pre-existing library has a**
  - procedural implementation of a specific integration method

```
REAL(rk) FUNCTION integ_qag(a, b, func, param)
```

  - with a procedure argument

QAG: „adaptive integration“

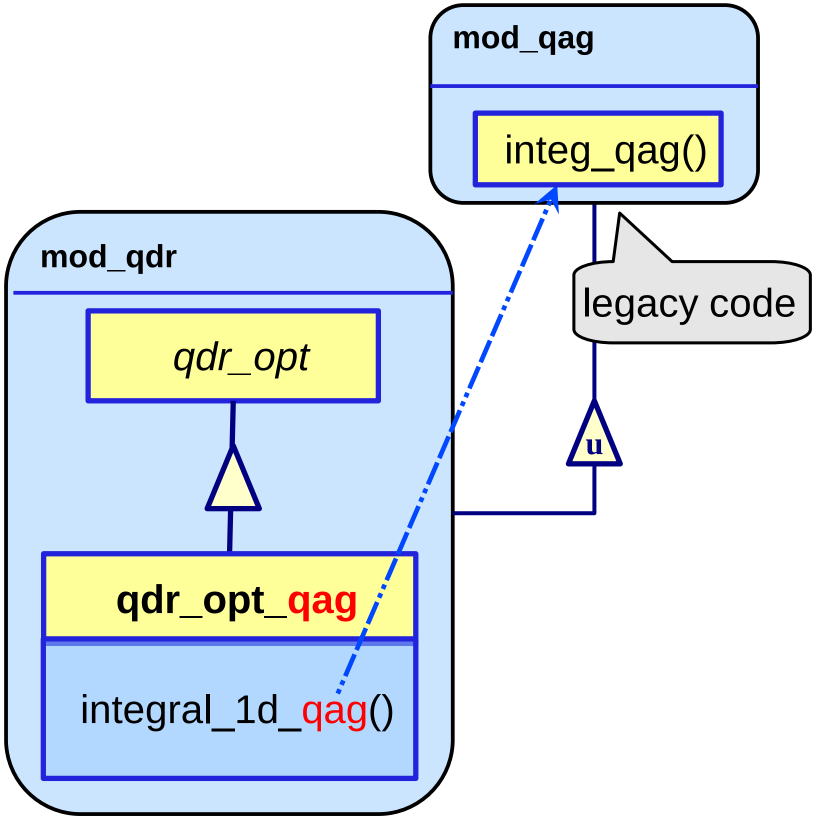
```
REAL (rk) FUNCTION func(x, param)
  REAL(rk), INTENT(IN) :: x
  TYPE(param_qag), INTENT(IN) :: param
END FUNCTION
```

  - and an appropriate type `param_qag`, all defined inside a module **`mod_qag`**

- **Target: re-use this library code**
  - as one more variant in the strategy pattern

Adapting legacy code (2)

- **Transition from old interface to new interface:**
  - concept is called “**class Adapter**” or “**Wrapper**”
  - old interface only used in **`mod_qdr`** module, invisible to client



- **Notes:**
  - C++ Adapter uses class for implementation inheritance (multiple inheritance required)
  - Fortran can exploit use association as secondary inheritance mechanism

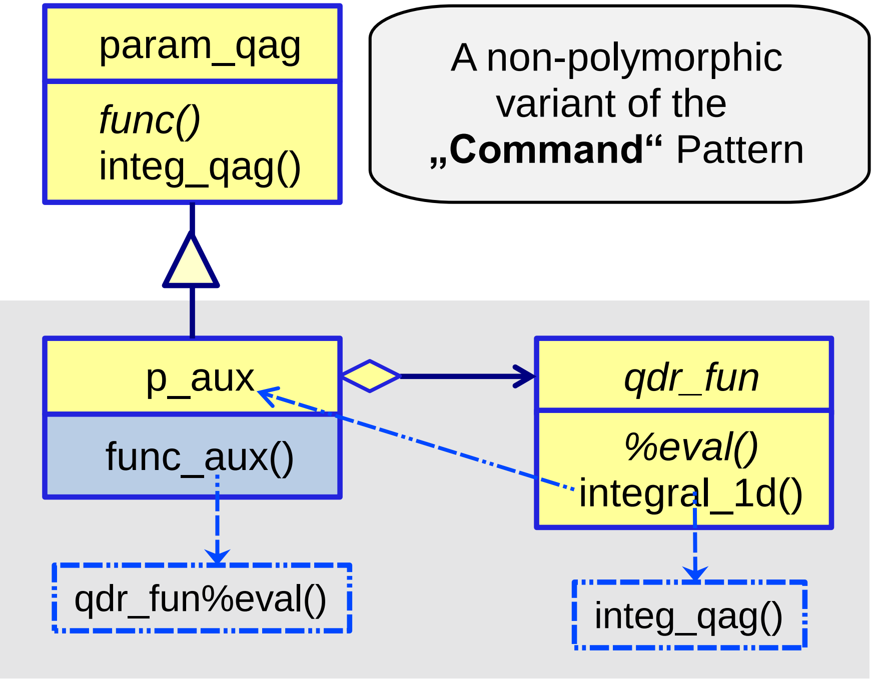
Adapting legacy code (3):
Solving the argument function signature mismatch

- **Procedure:**

  1. create an auxiliary function with the legacy signature that can be used as an argument for the internal invocation of **`integ_qag()`**
  2. make an object of type **`qdr_fun`** available inside that function

  → this requires defining an auxiliary type as an extension of **`param_qag`**

  → only one object of that type will be needed



```
TYPE, EXTENDS(param_qag) :: p_aux
  CLASS(qdr_fun), POINTER :: &
                    p => null()
END TYPE
```

Implementation of auxiliary function func_aux()

A module procedure in the module mod_qdr

```
REAL(rk) FUNCTION func_aux(x, pr)
  REAL(rk), INTENT(IN) :: x

  CLASS(param_qag), INTENT(IN) :: pr
  SELECT TYPE (pr)
    TYPE IS (p_aux)
! unpack qdr_fun object
    func_aux = pr%p%eval(x)
  END SELECT
END FUNCTION
```

declared type of the actual argument that matches **pr**

only one type is possible here

assume pr%p is associated

For consistency, one change is needed in mod_qag

```
TYPE(param_qag) → CLASS(param_qag)
```

in interface declaration of argument function

- semantics remain identical, but **recompilation** is needed

Implementation of integral_1d_qag()

- A module procedure in the module mod_qdr

```fortran
REAL(rk) FUNCTION integral_1d_qag( intv, fun, status)
  REAL(kind=rk), INTENT(IN) :: intv(2)
  CLASS(qdr_fun), INTENT(IN), TARGET :: fun
  INTEGER, OPTIONAL, INTENT(OUT) :: status

  INTEGER :: st_loc
  TYPE(p_aux) :: pr

  pr%param_qag = …
! register qdr_fun object with pr and
! call legacy routine
  pr%p => fun
  integral_1d_qag = integ_qag( intv(1),intv(2), &
                               func_aux, pr, st_loc)

  IF (present(status)) status = st_loc
END FUNCTION
```

this is the only object of type **p_aux**

supply base type object with needed information (only if needed outside invoked argument function)

following now: Exercise session 4