Advanced Fortran Topics

Partly a PRACE Event

Reinhold Bader
Gilbert Brietzke
Nisarg Patel

Leibniz Supercomputing Centre Munich
Fortran features under consideration

Continuing Standardization process:

<table>
<thead>
<tr>
<th>Version</th>
<th>Revision Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 66</td>
<td>ancient</td>
</tr>
<tr>
<td>Fortran 77 (1980)</td>
<td>traditional</td>
</tr>
<tr>
<td>Fortran 90 (1991)</td>
<td>large revision</td>
</tr>
<tr>
<td>Fortran 95 (1997)</td>
<td>small revision</td>
</tr>
<tr>
<td><strong>Fortran 2003</strong> (2004)</td>
<td>large revision</td>
</tr>
<tr>
<td><strong>Fortran 2008</strong> (2010)</td>
<td>mid-size revision</td>
</tr>
<tr>
<td><strong>TS 29113</strong> (2012)</td>
<td>extends C interop</td>
</tr>
<tr>
<td><strong>TS 18508</strong> (2015)</td>
<td>extends parallelism</td>
</tr>
<tr>
<td>Fortran 2018 (2018)</td>
<td>current standard</td>
</tr>
</tbody>
</table>

Focus of this course is on **F03** and **F08** and some **F18** features will be also covered.
Overview of covered features

**Day 1:**
- recapitulation of important features; object-based programming. First steps into object orientation

**Day 2:**
- further object-oriented features, I/O extensions, IEEE FP processing

**Day 3:**
- generic features, interoperation with C

**Day 4 – „Design Patterns“**
- how to use the OO features; first intro to PGAS programming

**Day 5 – „PGAS“:**
- parallel programming with coarrays

**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

- **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**

- **Guide to Fortran 2008 programming (introductory text)**

- **Modern Fortran – Style and Usage (best practices guide)**

- **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**

- **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/Materials+-+Advanced+Fortran+Topics](https://doku.lrz.de/display/PUBLIC/Materials+-+Advanced+Fortran+Topics)
Recapitulation:
Module features
Global variables
The environment problem

Some features from Fortran 2003
What is a module?

- A program unit
- ... that permits packaging of
  - procedure **interfaces**
  - **global** variables
  - named constants
  - **type definitions**
  - named interfaces
  - procedure implementations
- for reuse,
- ... as well as supporting
  - **information hiding**
  - (limited) **namespace** management

**Module definition syntax**

```
module <module-name>
  [ specification-part ]
  contains
  [ module-subprogram, ...]
end module <module-name>
```

Executable statements in a module can only appear in module subprograms

Also known as **encapsulation**
Illustrative example: heat conduction in 2 dimensions

**Simplest case:**
- partial differential equation for temperature $\Phi(x, y, t)$
  $$\frac{\partial \Phi}{\partial t} = \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2}$$
- produce stationary solution on a (unit) square
- provided: *initial* values inside; *boundary* values on NSWE edges

**Numeric model:**
- discretize square: increments $dx, dy$
- real array $\phi(:, :, :)$ of rank 2 models temperature field
  $\rightarrow$ use a *global* variable for this
- iteration process (Jacobi) based on 5 point stencil generates stationary solution
  $$\delta \Phi = \delta t \cdot [\Delta \Phi]_{\text{discretized}}$$
- with given time increment $\delta t$

© 2009-19 LRZ
Advanced Fortran Topics - LRZ section
Heat conduction: Global variable declarations

```fortran
module mod_heat
  implicit none
  private
  public :: heat_ival, heat_bval, heat_iter

  integer, parameter, public :: ndim = 200
  integer, parameter, public :: dk = ...

  real(dk) :: phi (ndim,ndim)
  real(dk) :: phinew (2:ndim-1,2:ndim-1)

contains
  ! implementation of module
  ! procedures - see later slides
end module mod_heat
```

**Call sequence:**
- set initial and boundary values
- repeatedly iterate until convergence
- print out result
Module procedures for the heat fields (1)

- **Perform iteration steps**

```fortran
real(dk) function heat_iter (dt, num)
  real(dk), intent(in) :: dt
  integer, intent(in) :: num
  real(dk) :: dphimax, dphi
  do it = 1, num
    dphimax = 0.0_dk
    do j = 2, size(phi,2) - 1
      do i = 2, size(phi,1) - 1
        dphi = dt * (…)
        phi_new(i, j) = phi(i, j) + dphi
        dphimax = max(dphi, dphimax)
      end do
    end do
    phi(2:size(phi,1)-1, &
    2:size(phi,2)-1) = phi_new
  end do
  heat_iter = dphimax
end function
```

- preserves boundary values

**Notes:**

- `heat_iter` is a module function in `mod_heat` → it has access to fields `phi`, `phinew` by *host association*
- global variables declared in a module are persistent: they implicitly have the *SAVE* attribute
- example code provided as basis for future exercises

© 2009-19 LRZ
Advanced Fortran Topics - LRZ section
Module procedures for the heat fields (2):
Procedure arguments

Example's boundary and initial value conditions:
- provided via functions
- a function or subroutine can be a dummy argument

Abstract interface
- in specification part of module

Initial value settings
- module procedure heat_ival in mod_heat

```
subroutine heat_ival(fival)
  procedure(f2) :: fival
  integer :: i, j
  real(dk) :: x, y
  do j=2, size(phi,2) - 1
    do i=2, size(phi,1) - 1
      ! calculate x, y
      phi(i, j) = fival(x, y)
    end do
  end do
end subroutine
```

(Legacy) Alternative
- replace procedure statement by a (regular) interface block

```
abstract interface
  real(dk) function f2(x, y)
  import :: dk
  real(dk), intent(in) :: x, y
end function
end interface
```

- describes interface of a procedure that does not (yet) exist
Heat main program

Compiles call sequence

Implements call sequence

Graphical representation

Program heat
use mod_heat
use mystuff
implicit none
integer :: it
real(dk), parameter :: &
  eps = 1e-6_dk
real(dk) :: dt
phi(3,4) = 0.0_dk
call heat_ival(myfun)
call heat_bval(...) do
dt = ... ! time step
  if (heat_iter(dt, 1) < eps) exit
end do
: ! print results
end program

mod_heat

phi(;;)
phinew(;;)
ndim = 200
dk = ...
heat_ival()
heat_bval()
heat_iter()
...

“provider”

heat

“client”

Use association provides an inheritance mechanism (for all
public entities of a module)
Some extensions for handling of globals

- Define entities which exist only once
  - example: temperature field

- Analogous for derived types:
  - client should not be able to create entity of a type

- type components can be public

- „Singleton“ programming pattern

Client usage:
- access public type components

```fortran
use mod_ptype

module mod_ptype
    implicit none
    private
    type, private :: ptype :
        ! type components
    end type
    type(ptype), public :: o_ptype
end module
```

```fortran
module mod_ptype
    implicit none
    private
    type, private :: ptype :
        ! type components
    end type
    type(ptype), public :: o_ptype
end module
```

- „Read-only“ objects:
  - attribute can not be applied to type definitions / components
  - client outside defining module shall not define object (neither directly or indirectly e.g., via a pointer)
  - for a pointer, PROTECTED refers to association status

Global entities: Threading issues

Typical threading model used
- OpenMP (assuming some knowledge here)
- directive based method for shared memory parallelism

Question discussed here:
- What happens if global variables need to be accessed from threaded parts of the code?
- How can „thread-safeness“ be achieved?
Example: counting objects

module mod_foo
  type :: foo
    private
    real, allocatable :: stuff(:)
  end type
  integer, protected, foo_count = 0
contains
subroutine foo_create (this, …)
  type(foo) :: this
  if (.not. allocated(this%stuff)) then
    ! allocate and initialize
    foo_count = foo_count + 1
  end if
end subroutine
subroutine foo_destroy (this)
  foo_count = foo_count - 1
end subroutine
end module mod_foo

C++ uses a static member variable

class Foo {
  public:
    Foo() : len_(0), stuff_(NULL) {};
    Foo(int, float *);
    :
  protected:
    static int count;
    int len_;
  private:
    float *stuff_; }

#include “Foo.h“
int Foo::count = 0;
Foo::Foo() {
  count += 1;
}
// same with all other constr.
// decrement in destructor

- module is encapsulation unit
- class is encapsulation unit
Updates to a shared entity

Example:
- execute object creation in parallel region

```fortran
type(foo) :: obj
!
! obj not created yet here
!
$omp parallel private(obj, ...) call foo_create(obj, ...) : ! do computations
call foo_destroy(obj)
$omp end parallel
!
! obj undefined
```

- beware definition status
- updates on `foo_count` are not thread-safe
  → inconsistencies / wrong values

Fix: use a named critical

```fortran
subroutine foo_create(this, ...) :
$omp critical (c_count)  
  foo_count = foo_count + 1
$omp end critical
end subroutine foo_create

subroutine foo_destroy(this) :
$omp critical (c_count)  
  foo_count = foo_count - 1
$omp end critical
end subroutine foo_destroy
```

- imagine `foo_count` is public → need an efficient tool to identify any problem

... and don't forget to switch on OpenMP everywhere!

© 2009-19 LRZ
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 \( \rightarrow \) 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:**

```fortran
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**

```fortran
double precision FUNCTION F (X)  
  double precision :: X
```

(user-provided function)

**Invocation:**

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

**Mass-production of integrals**

- may want to parallelize

```fortran
!$omp parallel do  
do i=istart, iend  
  : ! prepare  
    res(i) = d01ahf(..., my_fun, ...)
end do
!$omp end parallel do
```

- **need to check** library documentation: thread-safeness of d01ahf
Mismatch of user procedure implementation

- User function may look like this:

```fortran
subroutine user_proc(x, n, a, result)
  real(dk), intent(in) :: x, a
  integer, intent(in) :: n
  real(dk), intent(out) :: result
end subroutine
```

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

- or like this

```fortran
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

```fortran
module mod_user_fun
  double precision :: par
  integer :: n
contains
  function arg_fun(x) result(r)
    double precision :: r, x
    call user_proc(x, n, par, r)
  end function arg_fun
end module mod_user_fun
```

**Usage:**

```fortran
use mod_user_fun
par = ... ; n = ...
res = d01ahf(..., arg_fun, ...)
```
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:

```
!$omp parallel do
  do i=istart, iend
    par = ...; n = ...
    res(i) = d01ahf(..., arg_fun, ...)
  end do
!$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**

```fortran
module mod_user_fun
  double precision :: par
  integer :: n
  !$omp threadprivate (par, n)
  ...
end module
```

Usage may require additional care as well

```fortran
par = ...
:!$omp parallel do copyin(par)
  do i = istart, iend
    n = ...
    ... = d01ahf(..., arg_fun, ...)
    if (...) par = ...
  end do
:!$omp end parallel do
```

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

preparation step:
- set baseline parameters (a, b, p)
- produce first argument x

calculate f(x,p)
- for requested x

solution iteration step:
- feed in function value
- obtain intermediate result, next argument x, and state

check state
- complete
- unfinished

done
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
```

- 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
Taking Solution 2 a step further

- **Disadvantage:**
  - iteration routine completes execution while algorithm still executes
  - this may cause a big memory allocation/deallocation overhead if it uses many (large) stack (or heap) variables with local scope

- **Note:** giving such variables the SAVE attribute causes the iteration routine to lose thread-safeness

- **Concept of „coroutine“**
  - type of procedure that can interrupt execution without deleting its local variables
  - co-routine may **return** (i.e. complete execution), or **suspend**
  - invocation may **call**, or **resume** the coroutine
  (implies rules about invocation sequence)
  - no language-level support for this exists in Fortran
  - however, it can be emulated using OpenMP
Coroutine emulation via OpenMP tasking

- Separate tasks are started for
  - supplier, and for
  - consumer of function values

```fortran
:  n = ...; par = ...; a = ...; b = ...; eps = ...  
  flag = flag_need_iter
!$omp parallel num_threads(2) proc_bind(master)
!$omp single
!$omp task ...
    do
      call user_proc(x, n, par, fval)
    end do
!$omp end task
!$omp task
    call integrate_c(a, b, eps, fval, x, &
                   result, flag)
!$omp end task
!$omp end single
!$omp end parallel
:  
```

- Explicit synchronization needed
  - between supplier and consumer
  - functional (vs. performance) threading
  - involved objects: x, fval
  - use an integer flag for synchronization

```
t1                      t2
block until flag==1
produces x and sets flag:=1
block until flag==0
consume f(x,...)
calculate f(x, ...)
and set flag := 0
```

continues executing until the algorithm has completed
Synchronization code

Look at task block „t1“ from previous slide in more detail:

```fortran
!$omp task private(flag_local)
!$omp taskyield
   iter: do
      spin: do
         !$omp atomic read
            flag_local = flag
            if (flag_local == flag_need_fval) exit spin
            if (flag_local > 1) exit iter
         end do spin
$omp flush(x)
   call user_proc(x, n, par, fval)
$omp flush (fval)
$omp atomic write
   flag = flag_need_iter
$omp taskyield
   end do iter
$omp end task
```

A mirror image of this is done inside `integrate_c()`.

Grey area with respect to Fortran conformance (aliasing rules) the `TARGET` attribute might help.
Dynamic memory and object-based design
Recapitulation: dynamic objects

- **Add a suitable attribute to an entity:**
  - initial state is „unallocated“
  - initial state is "unassociated"
  - deferred shape
  - use of heap memory

- **Typical life cycle management:**
  - `allocate(x(2:n), p(3), stat=my_status)`
  - `x(:) = ...`
  - `p(:) = ...`
  - definitions and references
  - `deallocate(x, p)`

- **Status checking:**
  - `if (allocated(x)) then; ...`
  - `if (associated(p)) then; ...`
  - logical functions

implified lower bounds are possible (use LBOUND and UBOUND intrinsics)
**ALLOCATABLE vs. POINTER**

- **An allocated allocatable entity**
  - is an object in its own right
  - becomes auto-deallocated once going out of scope
- **An associated pointer entity**
  - is an **alias** for another object, its **target**
  - all definitions and references are to the target
  
  ```fortran
  allocate(p(3), stat=my_status)
  p(:) = ...
  deallocate(p)
  p => tg; p(2) = 2.0
  nullify(p) or p => null()
  
  real, target :: tg(3) = 0.0
  ```

- undefined (third) state should be avoided
Implications of POINTER aliasing

- Multiple pointers may point to the same target
  
  ```fortran
  allocate(p1(n))
  p2 => p1; p3 => p2
  ```

- Avoid dangling pointers
  
  ```fortran
  deallocate(p2)
  nullify(p1, p3)
  ```

- Not permitted: deallocation of allocatable target via a pointer
  
  ```fortran
  real, allocatable, target :: t(:)
  real, pointer :: p(:)
  ```
Allocatable entities

- Scalars permitted:
  \[
  \text{real, allocatable :: } s
  \]

- LHS auto-(re)allocation on assignment:
  \[
  x = p(2:m-2)
  \]
  conformance LHS/RHS guaranteed

- The MOVE_ALLOC intrinsic:
  \[
  \text{call MOVE_ALLOC(from, to)}
  \]
  avoid data movement

Pointer entities

- rank changing „=>“:
  \[
  \text{real, target :: m(n)}
  \text{real, pointer :: p(:,:,)}
  \text{p(1:k1,1:k2) => m}
  \]
  rank of target must be 1

- bounds changing „=>“:
  \[
  \text{p(4:) => m}
  \]
  bounds remapped via lower bounds spec

Deferred-length strings:

- \[
  \text{character(len=:)}, \text{ allocatable :: } \text{ var_string}
  \]
- \[
  \text{var_string} = '\text{String of any length}'
  \]
  pointer also permitted, but subsequent use is then different
  LHS is (re)allocated to correct length
Container types (1)

Allocatable type components

```
type :: polynomial
  private
  real, allocatable :: f(:)
end type
```

- a "value" container

```
type :: sparse
  private
  integer :: index
  real :: value
  type (sparse), pointer :: next => null()
end type
```

- a "reference" container

- example type is self-referential → "linked list"
Containers types (2):
Object declaration and assignment semantics

Allocatable type components

- 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

- Assignment statement is equivalent to

```
s2%index = s1%index
s2%value = s1%value
s2%next => s1%next
```

„shallow copy“

© 2009-19 LRZ

Advanced Fortran Topics - LRZ section
**Container types (3): Structure constructor**

### Allocatable type components

- **type** (polynomial) :: p1

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

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

### POINTER type components

- **type** (sparse) :: s1

  ```fortran
  s1 = sparse( 3, 1.0, null() )
  ```

  - **alternative:**

    ```fortran
    s1 = sparse( 3, 1.0, t1 )
    ```

  - **not permitted:**

    ```fortran
    s1 = sparse( 3, 1.0, t2 )
    ```

  a constant cannot be a target

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

**When object becomes undefined**

- allocatable components are automatically deallocated

**usually will not** happen for POINTER components
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]  )
```

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

Type component of array element is a descriptor that references a memory area “elsewhere”
Allocatable and POINTER dummy arguments
(explicit interface required)

Allocatable dummy argument

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

```fortran
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
```

POINTER dummy argument

- example: handling of a „reference container“

```fortran
subroutine add_reference( a_container, item )
  type(container), intent(inout) :: a_container
  real, pointer, intent(in) :: item(:)
  if (associated(item)) a_container % item => item
end subroutine add_reference
```

Actual argument must have matching attribute

© 2009-19 LRZ
**INTENT semantics for dynamic objects**

<table>
<thead>
<tr>
<th>specified intent</th>
<th>allocatable dummy object</th>
<th>pointer dummy object</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong></td>
<td>procedure must not modify argument or change its allocation status</td>
<td>procedure must not change association status of object</td>
</tr>
<tr>
<td><strong>out</strong></td>
<td>argument becomes <strong>deallocated</strong> on entry</td>
<td>pointer becomes <strong>undefined</strong> on entry</td>
</tr>
<tr>
<td><strong>inout</strong></td>
<td>retains allocation and definition status on entry</td>
<td>retains association and definition status on entry</td>
</tr>
</tbody>
</table>

**„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(OUT) and default initialized types

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

```fortran
type :: person
   character(len=32) :: name = 'no_one'
   integer :: age = 0
end type

subroutine modify_person(this)
    type(person), intent(out) :: this :
    this%name = 'Dietrich'
    ! this%age is not defined
end subroutine
```

then, after invocation of

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:

```fortran
real, pointer :: my_item(:) => null
type(container), intent(inout) :: my_container(ndim)
allocate(my_item(-3:8))
call add_reference(my_container(j), my_item)
```

What arrives inside `add_reference`?

```fortran
subroutine add_reference(
    a_container%item => item
)
    if (associated(item))
```

- this is different from assumed-shape, where bounds are remapped
- it applies for both POINTER and ALLOCATABLE dummy objects
The CONTIGUOUS attribute can be specified for pointers

- (and also for assumed-shape arrays)
- difference to assumed-shape: programmer is responsible for guaranteeing the contiguity of the target in a pointer assignment

**Example:**

- also illustrates rank changing:

```fortran
real, pointer, contiguous :: matrix(:, :) :
allocate(storage(n*n))
matrix(lb:ub,lb:ub) => storage
```

- **matrix** can be declared contiguous because whole allocated array **storage** is contiguous
- if contiguity of target is not known, check via intrinsic:

```fortran
if ( is_contiguous(other_storage) ) then
    matrix(lb:ub,lb:ub) => other_storage
else
    ... with possibly new values for lb, ub
```

© 2009-19 LRZ

Advanced Fortran Topics - LRZ section
 Allocatable function results
( explicit interface required )

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

Possible invocations:
- efficient (uses auto-allocation on assignment):

```
integer, allocatable :: res(:)
res = deduplicate(array)
```

- less efficient (two function calls needed):

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

- function result is auto-deallocated after completion of invocation

```fortran
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
```
POINTER function results
(explicit interface required)

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)

Example: filtering a list

```fortran
function next_uppertr(s, i) result(r)
  type(sparse), target, intent(in) :: s
  integer, intent(in) :: i
  type(sparse), pointer :: r
  r => s
  do while (...)
    r => r%next
  end do
  if (...) r => null()
end function next_uppertr
```

invocation:

```fortran
type(sparse), target :: trm(nd)
type(sparse), pointer :: entry

set up my_matrix (linked list)
do i=1, nd
  entry => trm(i)
  do while ( associated(entry) )
    entry => next_uppertr(entry, i)
  end do
```

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

© 2009-19 LRZ

Advanced Fortran Topics - LRZ section

46
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)
  - especially avoid using functions with pointer result

- A few scenarios where pointers may not be avoidable:
  - information structures → program these in an encapsulated manner: user of the facilities should not see a pointer at all, and should not need to declare entities targets.
  - subobject referencing (arrays and derived types) → performance impact!
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)

Potential restrictions on signatures of specific procedures

- operators: functions with two arguments (one for unary operations)
- assignment: subroutine with two arguments
- overloaded structure constructor: function with type name as result
- user-defined derived type I/O (treated on day 2)

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
Generalizing generic interface blocks

Can be replaced by

```fortran
interface foo_generic
  module procedure foo_1
  module procedure foo_2
end interface
```

with generalized functionality:

Referenced procedures can be

- external procedures
- dummy procedures
- procedure pointers

Example:

```fortran
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
```

```fortran
interface bar_gen
  procedure foo
  end procedure
end interface
```

- is valid in F03
- is non-conforming if a `module procedure` statement is used
Represent sparse matrix

- `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

Occupancy graph

- non-zero elements represented by black dots

Creating, copying and operations of such objects

- topics for the next slides and the exercises
Overloading the structure constructor

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

```
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
```
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)
Applying default assignment properties

For the overloaded constructor, ...

\[
\text{type(sparse), allocatable :: A(:) :}
\]
\[
\text{allocate A}
\]
\[
A(i) = \text{sparse(colidx, values)}
\]

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

However, for a "regular" assignment,

\[
\text{type(sparse), allocatable :: A(:), B(:) :}
\]
\[
\text{allocate A}
\]
\[
A(i) = \text{sparse(colidx, values)}
\]
\[
B = A
\]

- \( B \) 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:

```fortran
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
```

• 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)

```fortran
A(i) = sparse(...)  
```  
- deep copy of component  
- becomes orphaned → potential leak  

**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)

```fortran
type :: sparse  
  :  
contains  
  final :: finalize_sparse  
end type
```

© 2009-19 LRZ
elemental recursive subroutine finalize_sparse(this)
  type(sparse), intent(inout) :: this
  if (associated(this%next)) then
    deallocate(this%next)
  end if
end subroutine

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 made on day 4**

following now: Exercise session 2
Recall aliasing of dummy arguments

**Definition**
- access to object (or sub-object) via a name other than the argument's name:
  1. (sub)object of actual argument is associated with another actual argument (or part of it)
  2. actual argument is (part of) a global variable which is accessed by name
  3. actual variable (or part of it) can be accessed by host association in the executed procedure

**Simplest example:**
- illustrates item 1

```fortran
subroutine foo(a, b)
  real, intent(inout) :: a
  real, intent(in) :: b
  a = 2 * b
  ... = b
  a = a + b
end subroutine
```

**some invocations:**
```fortran
real :: x, y
x = 2.0; y = x
call foo(x, x)  ! aliased – non-conforming
call foo(x, y)  ! not aliased – x is 6.0
call foo(x, (x))  ! not aliased – x is 6.0
```
Aliasing restrictions

Restriction 1:
- if (a subobject of) the argument is defined in the subprogram, it may not be referenced or defined by any entity aliased to that argument

Notes:
- this restriction renders the first call illegal
- but aliasing is not generally disallowed
- exceptions to this rule will be discussed later

Intent:
- enable performance optimizations by statement reordering and/or register use
- avoid ambiguities in assignments to dummy arguments

Restriction 2:
- changes of allocation or association status of (part of) a dummy argument may only be performed via this argument.
- subsequent to such a change, any references or definitions of the object may be only via this argument

Note:
- deals with expected semantics of handling descriptor passing for allocatable or POINTER objects (usually copy-in/out)

Diagnosis of aliasing:
Requires inspection of procedure implementation as well as its invocation
- invocation for whether aliasing occurs
- implementation for whether the restrictions are violated
A non-conforming variation on the factory method seen earlier

```fortran
program simulation
  implicit none
  real, allocatable :: field(:,:,,:)
  call read_simulation_data(field, 'my_f.dat')
contains
  subroutine read_simulation_data(simulation_field, file_name)
    real, allocatable, intent(out) :: simulation_field(:,:,,:)
    character(len=*), intent(in) :: file_name
    allocate(simulation_field(:,:,,:)) ! and fill in values
    if (.not. allocated(field)) &
      allocate(field(:,:,,:)) ! and possibly give it values
  end subroutine read_simulation_data
end program
```

- and after return from the procedure further bad effects will occur
- interdiction against this applies for ALLOCATABLE and POINTER objects

copy in/out of descriptor for field might be done
argument association
this statement may well get executed and succeed
Exceptions to Restriction 1: POINTER dummy arguments

- By definition, pointers implement aliasing to their target
  - hence, for a dummy argument with the POINTER attribute restriction 1 does not hold
    (a pointer can be regarded as an orphaned dummy argument)

- Note that:
  - the aliasing property implies that the POINTER attribute has a negative performance impact;
  - the TARGET attribute on an object indicates to the compiler that pointers may be associated with the object or part of it. Optimization may depend on whether this currently is the case.
Example for permitted aliasing

The following program is conforming

```fortran
program aliasing_1
    implicit none
    real, pointer :: p(:)
    allocate (p(-10:10))
    call modify_ptr(p)
    deallocate(p)
contains
    subroutine modify_ptr(x)
        real, pointer, intent(in) :: x(:)
        integer :: i
        do i = lbound(p,1) + 1, ubound(p,1)
            p(i) = p(i) + x(i - 1)
        end do
    end subroutine modify_ptr
end program
```

- note the explicit interface

both actual and dummy argument have the POINTER attribute

p is argument associated with x

compiler cannot vectorize this code (effectively, a flow dependency)

p is host associated
Exceptions to Restriction 1: Dummy arguments with the TARGET attribute

Restriction 1 is lifted under the following additional conditions:

1. Dummy argument
   - is not `intent(in)` or `value`
   - is a scalar or an assumed-shape array

2. Actual argument
   - also has the TARGET attribute
   - is not an array section with a vector subscript

These conditions

- suppress copy-in/out and
- preserve pointer association across the interface (if the ultimate actual argument has the POINTER attribute, or a global pointer is associated with the dummy argument)
Second example for permitted aliasing

- The following program is conforming

```fortran
program aliasing_2
  implicit none
  real, pointer :: p(:)
  allocate (p(-10:10))
  call modify_tgt(p)
  deallocate(p)
contains
  subroutine modify_tgt(x)
    real, target, intent(inout) :: x(:)
    integer :: i, ip
    do i = 2, size(x, 1)
      ip = i + lbound(p,1) - 1
      p(ip) = p(ip) + x(i - 1)
    end do
  end subroutine modify_tgt
end program
```

- note the explicit interface
The following program is non-conforming

```fortran
program aliasing_3
  implicit none
  real, target :: q(20)
  real, pointer :: p(:)
  call modify_bad(p, size(p,1))
contains
  subroutine modify_bad(x, n)
    real, target, intent(inout) :: x(*)
    integer, intent(in) :: n
    integer :: i
    do i = 2, n
      p(i) = p(i) + x(i - 1)
    end do
  end subroutine modify_bad
end program
```

- inside the procedure, `associated(p, x)` may return false or true
Generally, an **explicit** interface is required

- for having the TARGET attribute on a dummy argument

**Case 1:**

dummy argument has the attribute, but actual does not

- then, any pointer associated with the target during execution of the subprogram becomes undefined at its completion
- association is with the dummy argument only, not the actual argument

**Case 2:**

actual argument has the attribute, but dummy argument does not

- then, pointer associations with the actual argument are not affected
- but association with dummy argument is undefined

**Case 3:**

(Example on previous slide)

- if the additional conditions for aliasing permission are **not** fulfilled, pointer association is **not** guaranteed to be preserved across the invocation/completion
### Fortran POINTERs: Handling the argument association

<table>
<thead>
<tr>
<th>Actual Argument</th>
<th>Dummy Argument</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>object</strong></td>
<td>object</td>
<td>pointer assoc. with dummy argument becomes <strong>undefined</strong> on return</td>
</tr>
<tr>
<td><strong>pointer</strong></td>
<td><strong>not allowed</strong></td>
<td>pointer assoc. with dummy arg. is <strong>preserved</strong>. copy-in/copy-out not allowed for scalar and assumed shape array dummies</td>
</tr>
<tr>
<td><strong>target</strong></td>
<td></td>
<td>permitted in <strong>F08</strong> if dummy is intent(in)</td>
</tr>
</tbody>
</table>

- **object**: usually by reference, may need copy-in/copy-out (efficiency), no-alias assumption
- **pointer**: must be associated, dereference to target, may need copy-in/copy-out (efficiency)
- **target**: usually by reference, copy-in/copy-out allowed, no-alias assumption
An alternative aliasing mechanism

- **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
Example:

- given the type definitions and object declaration:

```fortran
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( 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
```

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

associate name
selector
q must not be defined
Recommendations for library design (1)

- **Library not a static entity**
  - may want to add to functionality
  - may need to fix bugs / design problems

- **Open/Closed principle (OCP)**
  - Any software entity should be
    - open for extension
    - closed against modification
  - client using the library should not run into trouble
  - at minimum, client **source code** should build against updated library and execute as prior to the update

- **Assumption for the following discussion:**
  - all library code is implemented in form of modules
    (modules have improved support for many aspects of software engineering)

- higher level of "closed": **binary compatibility** - either replacement of shared libraries or relinking is sufficient
Recommendations for library design (2)

- Changes to implementations
  - typically bug fixes in bodies of module procedures
  - interfaces (procedure signature) unchanged

- Consequences:
  - theoretically, relinking against the library should be sufficient
  - if compilation of client code is performed, recompilation of all units directly or indirectly depending on the changed module must be done („cascade“)

- Changes to existing interfaces
  - normally forbidden
  - may be able to circumvent incompatibility via introduction of a generic
    ```
    interface my_sub
      module procedure my_sub
      module procedure my_corrected_sub
    end interface
    ```
    (or an optional argument)
    - add `my_corrected_sub()` as a module procedure

- Consequence:
  - need to recompile all dependent clients and relink
Recommendations for library design (3)

- **Derived types**
  - keep type components *private* („information hiding“)
  - exposed type components cannot be changed → would typically render client code unworking, therefore violates the OCP

- **Global data**
  - Declaration (name, most attributes) cannot be changed if public (for the same reason)

- **Consequence of changes on private type components or global data**
  - need to recompile all dependent clients and relink

**following now: Exercise session 3**
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 may be (very) 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**

- mod_date
  - date
    - year, mon, day
  - datetime
    - hr, min, sec

**Fortran type extension**

```fortran
module mod_date
  type :: date
    private
    integer :: year = 0
  end type
date

  type, extends(date) :: datetime
    private
    integer :: mon = 0, day = 0
  end type
datetime

  type :: mod_date
    date, datetime
  end type
end module
```

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

**Prerequisite:**
- single inheritance only
- 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
```

- **Parent component**
  - `o_dt % date`

  - is an object of parent type
  - a subobject of `o_dt`
  - recursive references possible: `o_dt % date % day`

- **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`

- **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

  ![Inheritance Tree Diagram]
  - date ("base type")
  - date_calendar (i.e. Mayan, etc.)
  - datetime
  - datetime_zone
  - datetime_hires
  - etc.

- 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:
    ```fortran
    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
- 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

Example: a partially opaque derived type

```fortran
module mod_person
  type :: person
    private
    character(len=strmx) :: name
    integer :: age
    character(len=tmx), public :: location
  end type
end module ! module procedures are not shown

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
```

design decision: location is not encapsulated. Why?
Type extension (6): Structure constructor

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

    ```fortran
    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

    ```fortran
    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

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

```fortran
use mod_ext
type (datetime_hires) :: o_hires

o_hires = datetime_hires(msec=711)
```

because all other components will receive their default-initialized value

- 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

Declaration with CLASS:

\[
\text{class}(\text{date}), \ldots :: \text{o\_poly\_dt}
\]

- **declared** type is `date`
- **dynamic** type may vary at run time: may be declared type and all its (known) extensions (type compatibility)
- 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 memory
3. both of the above

Possible additional attributes:

- loosening of strict **F95** typing rules
- `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

Inheritance mechanism:
- actual argument ...
  - ... can be of declared type of dummy or an extension:

```fortran
subroutine inc_date(this, days)
  class(date), intent(inout) :: this
  real(rk), intent(in) :: days
  ! implementation → exercise
end subroutine
```

```fortran
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)

Argument association:
- dynamic type of actual argument is assumed by the dummy argument
Explain continued:

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

Restriction on use:

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

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

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

  ```fortran
  allocate(datetime :: ad, cd)
  ```

  becomes dynamic type

  ```fortran
  allocate(date :: bd(5))
  ```

  could omit since equal to base type

- pointer association

  ```fortran
  type(datetime_zone), &
  target :: t
  ...
  ! may need to deallocate cd
  cd => t
  ```

  dynamic type of cd is now datetime_zone
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) :: arr(10)
```

- arr(1)%p can have a dynamic type different from that of arr(2)%p
Polymorphism (6): Further allocation mechanisms

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
- array bounds are also transferred in sourced allocation

Molded allocation
- 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.

```
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:
- The `select type` construct must be polymorphic.

```
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,...
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.

Further `type guards` for:
- Other extensions

```
class default
  stop 'Type not recognized'
end select
```
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)

- **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

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

```fortran
class(date) :: o1
    ! initialize o1
select type (o1)
class is (datetime)
call inc_datetime(o1,.03_rk)
end select

class default
write(*,*) &
    'Cannot invoke inc_datetime on o1'
end select
```

[Diagram of inheritance tree showing part of the class hierarchy for dates and times, with class guard covering certain parts of the tree.]

© 2009-19 LRZ
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

  ```fortran
  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:

  ```fortran
  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:

- intrinsic types
- integer
- real
- date
- body
- etc.
- all extensible „base“ types

BIND(C) and sequence types
Polymorphism (10): UP pointer

- An UP pointer can point to anything:

```fortran
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 ...

```fortran
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)

- RTTI:

```fortran
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
```

- 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

```fortran
class(*), target :: o_up
type(...), pointer :: p_nonext
```

```fortran
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)`

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

- **Recommendation:**
  
  - 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

That's it for today.
Following now: Exercise session 4
Object-oriented programming (II)

Binding of procedures to Types and Objects
Motivation

- **Remember** `inc_date` and `inc_datatime` procedures:
  - programmer decides which of the two routines is invoked
  - for an object of dynamic type `date`, `inc_datatime` cannot be invoked

- **Suppose there is a desire to**
  - invoke incrementation depending on the **dynamic** type of the object:
    ```fortran
    class(date), allocatable :: o_d
    ```
    - `date`: `o_d%increment(...)` invokes `inc_date`
    - `datetime`: `o_d%increment(...)` invokes `inc_datatime`

- **This concept is also known as** dynamic (single) dispatch via the object
  - cannot use F95 style generics (polymorphism forces run-time decision)
Prolegomenon: Pointers to procedures (1)

**Declaration:**

```fortran
procedure(subr), pointer :: &
pr => null()
```

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

```fortran
interface
  subroutine subr(x)
    real, intent(inout) :: x
  end subroutine
end interface
```

**Usage:**

```fortran
real :: x
pr => subr
x = 3.0
call pr(x) ! invokes "subr"
```

**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

must associate before invocation
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()

pfun => fun
write(*,*) pfun(3.5)
returns fun(3.5)
pfun => sin
write(*,*) pfun(3.0)
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
```

**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
```
Procedures as type components

Two variants are supported:

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

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

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

Example:

```fortran
type :: data_send_container
    class(data), allocatable :: d
    procedure(send), &
    pointer :: send => null()
end type

type :: date
    ! previously defined comp.
    contains
    procedure :: &
    increment => inc_date
end type
```
Restrictions on the procedure interface

... apply for both variants

First dummy argument:

- declared type must be same type as the 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

```fortran
subroutine send(this, desc)
  class(data_send_container) :: this
  class(handle) :: desc
 : ! implementation not shown
end subroutine
```

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

- **Object-bound procedure**
  - mod_send
  - data_send_container
    - send()
  - send()

- **Type-bound procedure (TBP)**
  - mod_date
    - date
    - yr,mon,day
    - increment()
  - inc_date()

- 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

```fortran
type(data_send_container) :: c :
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
```

```fortran
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)
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:

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

```fortran
    type, extends(date) :: datetime :
    ! as before
    contains
      procedure :: increment => & inc_datetime
    end_type
```

with the binding above added,

```fortran
    call o_dt%increment(.03_rk)
```

- **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!

© 2009-19 LRZ  Advanced Fortran Topics - LRZ section 108
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

These cannot be overridden outside their defining module

```fortran
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, \( p_2 \) is not an overriding type-bound procedure, but a new binding that applies to all entities of \( \text{class}(t_2) \).
- \( p_2 \) therefore need not 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).
The **NON_OVERRIDABLE** attribute can be used in any binding.

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

```fortran
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)

- **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)

```fortran
type(datatime) :: dtt
call dtt%date%increment(120._rk)
```
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) ])
```

- **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

```fortran
type :: t1
  contains
  procedure, &
  pass(o1) :: pf
end type

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**

```fortran
type :: t3
  contains
    procedure, nopass :: pf
end type
```

- **Invocations:**

```fortran
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

Properties:

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

Example:

```
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)

**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
```

- 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
```

- 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

```fortran
module mod_file_handle
  use mod_handle
  type, extends(handle) :: file_handle
    private
    integer :: unit
  contains
    procedure :: open => file_open
  end type
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
```

*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

```fortran
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
```

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

Split off implementations (module procedures) into separate files

- Access is by host association (i.e. also to private entities)

Diagram:

- Module `m`
  - Procedure `()`
- Submodule `(m) s`
  - Procedure `()`
Submodule program units

Syntax

```fortran
submodule ( mymod ) smod_1
  : ! specifications
contains
  : ! implementations
end submodule
```

- Applies recursively: a descendant of `smod_1` is

```fortran
submodule ( mymod:smod_1 ) smod_2
  :
end submodule
```

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

Symbolic representation

```
mymod

h

smod_1

h

smod_2
```

© 2009-19 LRZ

Advanced Fortran Topics - LRZ section
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

```fortran
module mymod
  implicit none
  type :: t :
    end type :
end module
```

```fortran
submodule ( mymod ) smod_1
  type, extends(t) :: ts :
    end type :
  real, allocatable :: x(:,:,)
    : end submodule
end submodule
```
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 mod_date

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

```fortran
submodule (mod_date) date_procedures
  : ! specification part
contains
  module procedure write_date
    : ! implementation as shown before
  end procedure write_date
  module procedure create_date
    : ! implementation 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

```fortran
submodule (mod_date) date_procedures
  : ! specification part
  contains
    module subroutine write_date (this, fname)
      class(date), intent(in) :: this
      character(len=*) , intent(in) :: fname
    : ! implementation 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
    : ! implementation as shown before
    end function create_date
end submodule date_procedures
```

note syntactic difference to Variant 1
Dependency inversion explained

- **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

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

Beware:

- use association overrides host association → use of an **only** clause is advisable

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

**following now: Exercise session 5**
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
  - concept: 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

- `funds`
  - currency, amount
  - increment()

- `admin_funds`
  - interest
  - increment()

- `date`
  - day, mon, year
  - increment()

-how to define?
Declaring a generic type-bound procedure

**Starting point:**
- the type which first declares the binding that must be generic

```fortran
private
character(len=3) :: currency
real :: amount
contains
procedure, private :: &
  inc_funds
generic, public :: &
  increment => inc_funds
end type
```

**Adding specifics to a generic in a type extension:**
- 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

```fortran
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
```

**OCP:**
- may need to retrofit generic from simple TBP (easily done, at the cost of recompiling all clients)
Disambiguating procedure interfaces

Subroutine inc_funds:

```fortran
subroutine inc_funds(this, by)
  class(funds)
end subroutine inc_funds
```

Subroutine inc_date:

```fortran
subroutine inc_date(this, days)
  class(admin_funds)
end subroutine inc_date
```

Subroutine inc_both:

```fortran
subroutine inc_both(this, days, by)
  integer :: days    real :: by
end subroutine inc_both
```

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

```

```

```OFFAME

```

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/day2/multiple_inheritance

```

```
Overriding a specific binding in a generic TBP

Further type extension (in a different module)

```fortran
type, extends(admin_funds) :: &
  my_funds
: contains
  procedure :: &
    inc_both => inc_my_funds
end type
```

- with a module procedure:

```fortran
subroutine inc_my_funds(this, &
  ninc, by)
  class(my_funds) :: this
: ! ninc, by as before
end subroutine
```

Invocation:

```fortran
class(admin_funds), &
  allocatable :: o_mf

allocate(my_funds :: o_mf)
: ! initialize o_mf

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

Original binding `public` so it can be overridden.

Invokes overriding procedure `inc_my_funds` because dynamic type is `my_funds`.
Unnamed generic TBPs – defined operator

Example:

- unary trace operator

```fortran
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

```fortran
real function trace(this)
  class(matrix), intent(in) :: &
    this
end function
```

Invocation:

```fortran
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:

```fortran
type :: vector
  : ! see earlier definition
contains
  procedure :: plus1
  procedure :: plus2
  procedure, pass(v2) :: plus3
  generic, public :: &
    operator(+) => plus1, plus2, plus3
end type matrix
```

- Specifics:

```fortran
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

```fortran
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
Use italics to indicate generic-ness

- provide list of specific TBPs as usual
- overriding in subclasses can then be indicated as previously shown
Nonadvancing I/O
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:

  ```fortran
  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)', '(11)' – each square is 1 character (byte)

- ! real :: r; logical :: b
  read (22, ..., advance='NO') r

- after execution of

- ! real :: r; logical :: b
  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=iostats) ...
if(is_iostat_eor(iostats)) ...
```
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
Object-oriented
I/O Facilities:
User defined derived type I/O
I/O for derived data types

Non-trivial derived data type

type :: list
  character(len=:) , &
    allocatable :: name
  integer :: age
  type(list), pointer :: next
end type

Perform I/O using suitable module procedures

Disadvantages:

- recursive I/O disallowed
- I/O transfer not easily integrable into an I/O stream
  - defined by edit descriptor for intrinsic types and arrays
  - or sequence of binary I/O statement

- enables binding a subroutine to an I/O list item of derived type

  type(list) :: o_list
  ! set up o_list
  write(unit, fmt='(dt ...)', ...) &
    o_list

- example shows formatted output
- bound subroutine called automatically when edit descriptor DT is encountered
- other variants are enabled by using generic TBPs or generic interfaces
- can use recursion for hierarchical types
Binding I/O subroutines to derived types

- Interface of subroutines is fixed
  - with exception of the passed object dummy

- Define as special generic type bound procedure

```fortran
  type :: foo
    contains
      generic :: read(formatted) => rf1, rf2
      generic :: read(unformatted) => ru1, ru2
      generic :: write(formatted) => wf1, wf2
      generic :: write(unformatted) => wu1, wu2
  end type
```

- genericness refers to rank, kind parameters of passed object

- Define via interface block

```fortran
  interface read(formatted)
    module procedure rf1, rf2
  end module
end interface
```
DTIO module procedure interface
(dummy parameter list determined)

subroutine rf1(dtv,unit,iotype,v_list,iostat,iomsg)
subroutine wu1(dtv,unit,iostat,iomsg)

❑ dtv:
  • scalar of derived type
  • may be polymorphic
  • of suitable intent

❑ unit:
  • integer, intent(in) – describes I/O unit or 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)

- **Use of the statements**
  - open, close, rewind
  - backspace, endfile

- **File positioning:**
  - entry is left tab limit
  - no record termination on return
  - positioning with
    - rec=... (direct access) or
    - pos=... (stream access)

  is disallowed
Writing formatted output:
DT edit descriptor

Example:

```fortran
type(mydt) :: o_mydt ! formatted writing bound to mydt :
write(20, '(dt 'MyDT' (2, 10) )') o_mydt
```

- Available in `iotype` (empty string if omitted)
- Available in `v_list` (empty array if omitted)

Both `iotype` and `v_list` are available to the programmer of the I/O subroutine

- determine further parameters of I/O as programmer sees fit

Note:

- inside a formatted DTIO procedure („child I/O“), I/O is **nonadvancing**
  (no matter what you specify for ADVANCE)
Here: type definitions and DTIO-procedure implementations inside same module, e.g.:

```fortran
module mod_list
  type :: list
    integer :: age
    character(20) :: name
  type(list), pointer :: next
contains
  generic :: &
    write(formatted) => wl
end type list
contains
: ! to be continued

recursive subroutine wl &
  (this,unit,iotype,vlist,iostat,iomsg)
  class(list), intent(in) :: this
  integer , intent(in) :: unit, vlist(:)
  character(*) , intent(in) :: iotype
  integer, intent(out):: iostat
  character(*) :: iomsg
  ! .. Locals
  character(len=12) :: pfmt
  if (iotype /= 'DTList') return
  if (size(vlist) < 2) return
  ! internal IO to generate format descriptor
  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 ) return
  if (associated(this%next)) call wl &
    (this%next,unit,iotype,vlist,iostat,iomsg)
end subroutine
!:other implementations ...
end module
```
Example (cont'd): Client use

Client use formatted DTIO

type(list), pointer :: mylist
  ! : set up mylist
  ! : open formatted file to unit

write(unit, fmt='(dt "List" (4,20 ))', &
     iostat=is) mylist
  ! : open formatted file to unit

! : close unit and destroy list

Client use unformatted DTIO

type(mydt) :: o_mydt

! : unformatted writing (also) bound to mydt
! : open unformatted file to unit 21

write(21[, rec=...]) o_mydt

Final remarks: Unformatted DTIO

- bound subroutine
  - with shorter argument list

- is automatically invoked upon execution of write statement

- additional arguments (e.g. record number) only specifiable in parent data transfer statement
Asynchronous I/O
Basic concept

Flow diagram

- read A
- work on B
- wait
- work on A

Implementation

```fortran
real, dimension(100000) :: a, b
open(20,...,asynchronous='yes')

read(20,asynchronous='yes') a
! do work on something else
:
wait(20)
! do work with a
... = a(i)
```

- **Ordering requirements**
  - apply for a sequence of data transfer statements on the same I/O unit
  - but not for data transfers to different units
Conditions for asynchronous execution

Necessary conditions

- OPEN statement
  
  ```fortran
  open(unit=<number>, ..., asynchronous='yes')
  ```
  
  - prerequisite for performing asynchronous I/O statements on that unit

- READ or WRITE statements
  
  ```fortran
  [read|write](unit=<number>,..., &
  asynchronous='yes',id=<integer_tag>) <data_entity>
  ```
  
  - ID specifier allows to assign each individual statement a tag for subsequent use

Actual asynchronous execution

- is at processors discretion
- is most advantageous for large, unformatted transfers
The **WAIT** statement

- **Block until asynchronously started I/O statement has completed**

  ```fortran
  wait(unit=<number> [, &
       id=<integer_tag>, &
       end=<label>, &
       eor=<label>, &
       err=<label>, &
       iostat=<integer_status>])
  ```

- **Implication:**
  - can have multiple outstanding asynchronous I/O statements to the same unit

- **Implicit WAIT is incurred:**
  - by **BACKSPACE**
  - by **REWIND**
  - by **ENDFILE**
  - possibly by **INQUIRE**
  - by **CLOSE**

- **Orphaned WAIT**
  - with respect to unit or id
  - has no effect

- **if ID tag present → wait only for tagged statement**

- **else wait for all outstanding I/O statements on that unit**
Non-blocking execution

Option for check of I/O completion
- extension of INQUIRE statement

```fortran
inquire(unit=<number>, pending=<logical>, id=<integer_tag>)
```

- **PENDING** specifier returns `.true.` if operation tagged by ID is not yet complete
- if no ID present, all outstanding I/O statements must be complete
- **PENDING** specifier returns `.false.` if operation tagged by ID is complete
- refers to completion status of all outstanding I/O statements if no ID present
- a return value `.false.` implies a WAIT (i.e. an implementation may decide to wait for completion while the INQUIRE executes)
Affector entities

- Entity in a scoping unit
  - item in an I/O list
  - item in a NAMELIST
  - SIZE= specifier

- associated with an asynchronous I/O statement

- Constraints on affectors:
  - must not be redefined,
  - become undefined, or
  - have pointer association status changed

- while I/O operation on it is pending

- While asynchronous input is pending
  - affector must not be referenced or associated with a VALUE dummy argument
Recall prototypical case:

```fortran
open(20,...,asynchronous='yes')
...
read(20,asynchronous='yes') a :
! compiler may not prefetch "a" here
wait(20)
... = a(i)
```

- asynchronous I/O puts constraints on code movement by the compiler
- all affectors automatically acquire the **ASYNCRONOUS** attribute in the above case
  - once acquired in a scoping unit, will propagate
  - all subobjects of an affector also have the attribute
However consider

```fortran
subroutine read_async(unit,id,this)
  integer, intent(in) :: unit
  integer, intent(out) :: id
  type(...), intent(out) :: this
  read(unit=iu,id=id, &
    asynchronous='yes') this
end subroutine

subroutine work(unit,id,this)
  integer, intent(in) :: unit
  integer, intent(in) :: id
  type(...), intent(in), &
    asynchronous :: this
  wait(unit, id=id)
  ... = this
end subroutine
```

- need explicit attribute to suppress code motion

Consider further the call sequence

```fortran
type(...), asynchronous :: this
: ! Open unit
call read_async(unit,id,this)
:call work(unit,id,this)
... = this
```

- due to intent(in) in work() compiler could move loads of this before call to work()
- again, need explicit `ASYNCRONOUS` attribute
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 not (necessarily) 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 standard

- **What bandwidths are available?**
  - normal SCSI disks
    - ~100 MByte/s
  - NAS storage arrays at LRZ:
    - up to 2 GByte/s
  - SuperMUC storage arrays:
    - up to 150 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

---

### 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

- 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

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

- proper tuning
  - performance may exceed that for array sections

Large blocks, but wrong order. On some platforms this may give a performance hit
→ re-copy array or reorganize data
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
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
IEEE Arithmetic
and
IEEE Floating Point
Exception Handling
IEEE-754

- ISO-IEC standard for binary floating point processing

- Defines
  - floating point representations
  - prescriptions for conforming +,-,*,/,,sqrt (portable FP programming)
  - rounding modes
  - exceptions and exception handling mechanisms

  Standard CPUs have **hardware** support for (most of) the above

- Reference: David Goldberg's article
  - What Every Computer Scientist Should Know about Floating-Point Arithmetic

- Further information available at
  - [http://grouper.ieee.org/groups/754/](http://grouper.ieee.org/groups/754/)
Intrinsic modules (1)

**IEEE support in Fortran**
- three intrinsic modules → **subset** support

**IEEE features**
- contains a number of constants of type `ieee_feature_type`

```fortran
use, intrinsic :: &
  ieee_features, only : ieee_divide
```

- effect as for a compiler switch
- applies for complete scoping unit
- possible performance impact → use an **only** clause to limit effect
- if a feature unsupported → compilation fails
List of constants in `ieee_features`

**Default settings:**

- may be an arbitrary subset

<table>
<thead>
<tr>
<th>Named constant</th>
<th>Effect of access in scoping unit for at least one kind of real</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ieee_datatype</code></td>
<td>must provide IEEE arithmetic</td>
</tr>
<tr>
<td><code>ieee_denormal</code></td>
<td>must support denormalized numbers</td>
</tr>
<tr>
<td><code>ieee_divide</code></td>
<td>must support IEEE divide</td>
</tr>
<tr>
<td><code>ieee_halting</code></td>
<td>must support control of halting</td>
</tr>
<tr>
<td><code>ieee_inexact_flag</code></td>
<td>must support inexact exception</td>
</tr>
<tr>
<td><code>ieee_inf</code></td>
<td>must support $-\infty$ and $+\infty$</td>
</tr>
<tr>
<td><code>ieee_invalid_flag</code></td>
<td>must support invalid exception</td>
</tr>
<tr>
<td><code>ieee_nan</code></td>
<td>must support NaN</td>
</tr>
<tr>
<td><code>ieee_rounding</code></td>
<td>must support control of all four rounding modes</td>
</tr>
<tr>
<td><code>ieee_sqrt</code></td>
<td>must support IEEE square root</td>
</tr>
<tr>
<td><code>ieee_underflow_flag</code></td>
<td>must support underflow exception</td>
</tr>
</tbody>
</table>
Intrinsic modules (2)

- IEEE exceptions
  - definitions of types, flags and procedures for exception handling
  - note: no support for (user-defined) handler callback functions

- IEEE arithmetic
  - definitions of classes of floating point types
  - definitions of rounding modes; functions for setting and getting these modes

- Using `ieee_arithmetic` implies use of `ieee_exceptions`
Models for integer and real data

**Numeric models for integer and real data**

\[ i = s \times \sum_{k=1}^{q} w_k \times r^{k-1} \]

\[ x = b^e \times s \times \sum_{k=1}^{p} f_k \times b^{-k} \text{ or } x = 0 \]

**integer kind** is defined by

- positive integer \( q \) (digits)
- integer \( r > 1 \) (normally 2)

**integer value** is defined by

- sign \( s \in \{\pm 1\} \)
- sequence of \( w_k \in \{0, ..., r-1\} \)

**real kind** is defined by

- positive integers \( p \) (digits), \( b > 1 \) (base, normally \( b = 2 \))
- integers \( e_{\text{min}} < e_{\text{max}} \)

**real value** is defined by

- sign \( s \in \{\pm 1\} \)
- integer exponent \( e_{\text{min}} \leq e \leq e_{\text{max}} \)
- sequence of \( f_k \in \{0, ..., b-1\} \), \( f_1 \) nonzero

Integers are not dealt with through the IEEE facilities.

Base 2 \( \rightarrow \) "Bit Pattern"
Inquiry intrinsics for model parameters

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digits(x)</td>
<td>for real oder integer x, returns the number of digits (p, q respectively) as a default integer value.</td>
<td>minexponent(x), maxexponent(x)</td>
<td>for real x, returns the default integer $e_{\text{min}}$, $e_{\text{max}}$ respectively</td>
</tr>
<tr>
<td>precision(x)</td>
<td>for real or complex x, returns the default integer indicating the decimal precision (=decimal digits) for numbers with the kind of x.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>range(x)</td>
<td>for integer, real or complex x, returns the default integer indicating the decimal exponent range of the model x belongs to.</td>
<td>radix(x)</td>
<td>for real or integer x, returns the default integer that is the base ($b, r$ respectively) for the model x belongs to.</td>
</tr>
</tbody>
</table>
Inquiry intrinsics for model numbers

- **Example representation:** \( e \in \{-2, -1, 0, 1, 2\} \), p=3
  - look at first positive numbers (spacings \( \frac{1}{32}, \frac{1}{16}, \frac{1}{8} \) etc.)

\[ u = \frac{1}{8} \]

- largest representable number: \( \frac{7}{2} \)
  (beyond that: overflow)

\[ rrs\text{spacing}(x) = \frac{\text{abs}(x)}{\text{spacing}(x)} \]

\[ \text{tiny}(x) \]

\[ \text{spacing}(0.35) \]

\[ \text{nearest}(0.35, -1.0) \]

\[ \text{epsilon}(x) \]

\[ \text{huge}(x) \]

\[ \text{Mapping } \mathbb{R} \ni x \rightarrow fl(x) \]
  - to nearest model number
  - maximum relative error

\[ fl(x) = x \cdot (1 + d), |d| < u \]
... more realistic models

**Typically used representations:** IEEE-754 conforming

- matched to hardware capabilities

<table>
<thead>
<tr>
<th>real kind</th>
<th>dec. digits</th>
<th>base 2 digits</th>
<th>dec. exponent range</th>
<th>base 2 exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>6</td>
<td>24</td>
<td>$10^{-37} \ldots 10^{+38}$</td>
<td>-125 ... +128</td>
</tr>
<tr>
<td>extended</td>
<td>15</td>
<td>53</td>
<td>$10^{-307} \ldots 10^{+308}$</td>
<td>-1021 ... +1024</td>
</tr>
</tbody>
</table>

**Negative zero:**

- hardware may distinguish from positive zero
- e.g., rounding of negative result toward zero retains sign,
- e.g., I/O operations (sign stored in file)
Closure issues (1): Rounding

**Arithmetic operations:**
- result typically **outside** the model
- requires **rounding**
- implementation dependency, but all good ones adhere to „standard model“

\[
fl_{op}(x, y) = (x \ op \ y) \cdot (1 + d), \quad |d| \leq u; \ op = +, -, \ast, /.
\]

- precision achieved in IEEE algorithms by using guard digits

There exist relevant algorithms for which less strict models cause **failure**!

**Rounding modes:**
- modify exact result to become a representable number
- **nearest**: to nearest representable value (NRV)
- **to-zero**: go toward zero to NRV
- **up**: go toward \(+\infty\) to NRV
- **down**: go toward \(-\infty\) to NRV

**Note:**
- division \(a/b\) executed as \(a \ast (1/b)\) may not be IEEE conforming (roundoff)
- conversely: enforcing IEEE conformance may have performance impact
Closure issues (2): Rounding

- IEEE-754 rounding modes
  - all fulfill the model from the previous slide
- Named constants in module `ieee_arithmetic`
  - `ieee_nearest`
  - `ieee_to_zero`
  - `ieee_up`
  - `ieee_down`
  - `ieee_other`
  - all of type `ieee_round_type`

Ask for full rounding support:

```fortran
use, intrinsic :: ieee_features, only : ieee_rounding
```
Example program illustrating rounding

```
use, intrinsic :: ieee_arithmetic, only : ieee_nearest, ieee_up, &
& ieee_down, ieee_set_rounding_mode, ieee_support_rounding

real    :: a, d
integer :: i

real constant d = 0.232
if (ieee_support_rounding(ieee_XXX,a)) then
  write(*,fmt='("Round XXX:"')
  call ieee_set_rounding_mode(ieee_XXX)
  a = 1.5
  do i = 0, 4
    a = a / d
    write(*, fmt='(f13.6)') a
  end do
else
  write(*, fmt='("Rounding mode ieee_XXX unsupported")')
end if
```

Produce output:

- **Round nearest:**
  - 6.465518
  - 27.868610
  - 120.123322
  - 517.772949
  - 2231.780029

- **Round up:**
  - 6.465518
  - 27.868612
  - 120.123337
  - 517.773071
  - 2231.780762

- **Round down:**
  - 6.465517
  - 27.868608
  - 120.123314
  - 517.772888
  - 2231.779541
Control propagation of rounding error

- **Strategy 1:**
  - choose most appropriate rounding
  - randomized may be best, but is **expensive**!

- **Strategy 2: Interval arithmetic**
  - [a,b] (ordered interval in \( \mathbb{R} \))
  - \( \mu([a,b]) := b - a \)
  - define operations
    - \([a,b] + [c,d] := [a+c,b+d] \)
    - \([a,b] * [c,d] := [ac,bd] \) if \( a,c > 0 \) etc.
  - keep actual value within small interval
  - „small“ may become difficult:
    - \( \mu([a,b] *[c,d]) = d*\mu([a,b]) + a*\mu([c,d]) \)

- **Implementation of IA**
  - may want to use rounding to guarantee enclosure

- **Note:**
  - there exist multiple variants of IA
  - a standardization effort (outside Fortran) is under way

- **Interval software**
  - INTLIB library / **interval_arithmetic** Fortran module from R. Baker Kearfott's page at [http://interval.louisiana.edu/](http://interval.louisiana.edu/)
Obtaining the current rounding mode

Intrinsic module procedure:

```fortran
use, intrinsic :: &
  ieee_arithmetic, only : &
  ieee_nearest, ieee_round_type, &
  ieee_get_rounding_mode, &
  operator(==)

type(ieee_round_type) :: round_value

call ieee_get_rounding_mode( round_value )

if (round_value == ieee_nearest) &
  write(*,*), 'Round to nearest.'
```

Remember:

- rounding error and truncation error are two different things
- the latter usually arises from finite approximation of a representation of a function; explicit truncation error terms can sometimes (but not always) be established

- entities of **opaque** type

  `ieee_round_type`

- the only allowed operations

  are assignment, `==, /=`
Closure issues (3): special FP numbers

Three variants:

- $\infty$, $-\infty$

- NaN (signaling or quiet)

- Denormal numbers ($f_1 = 0$ and $e = e_{\min}$)

From which operations?

- $1.0 / 0.0$ (or $-1.0 / 0.0$), more general: numbers exceeding $\text{HUGE}(x)$ (or smaller than $-\text{HUGE}(x)$)

- $0.0 / 0.0$, $\text{SQRT}(-1.0)$, more general: invalid operations

- **gradual** underflow

Production of special FP numbers triggers exceptions
IEEE arithmetic:
Inquiry functions for real and complex types

<table>
<thead>
<tr>
<th>logical function specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee_support_datatype([x])</td>
<td>supports at least a subset of IEEE arithmetic (operations, rounding mode, kind, some intrinsics)</td>
</tr>
<tr>
<td>ieee_support_denormal([x])</td>
<td>supports IEEE denormalized numbers</td>
</tr>
<tr>
<td>ieee_support_divide([x])</td>
<td>supports IEEE conformant division</td>
</tr>
<tr>
<td>ieee_support_inf([x])</td>
<td>supports IEEE infinity facility</td>
</tr>
<tr>
<td>ieee_support_nan([x])</td>
<td>supports IEEE Not-a-Number facility</td>
</tr>
<tr>
<td>ieee_support_rounding(rd_value [,x])</td>
<td>supports specified rounding mode as well as setting via intrinsic. Additional argument is type(ieee_round_type), intent(in) :: rd_value</td>
</tr>
<tr>
<td>ieee_support_sqrt([x])</td>
<td>supports IEEE conformant square root</td>
</tr>
<tr>
<td>ieee_support_standard([x])</td>
<td>supports all IEEE facilities within ieee_arithmetic</td>
</tr>
<tr>
<td>ieee_support_underflow_control([x])</td>
<td>supports control of IEEE underflow mode via intrinsic</td>
</tr>
</tbody>
</table>

All functions return a result of type logical and take (at least) an optional argument of type real.
IEEE arithmetic:
 Underflow handling & real kinds

```fortran
subroutine ieee_get_underflow_mode(gradual)
  logical, intent(out) :: gradual
  .true. is returned of gradual underflow is in effect

subroutine ieee_set_underflow_mode(gradual)
  logical, intent(in) :: gradual
  .true. lets gradual underflow come into effect

integer function ieee_selected_real_kind([p] [,r])
  same functionality as selected_real_kind()
  but returns a kind value for which ieee_support_datatype(x) is true
```
The five exceptions defined in IEEE-754

- **Treated via (hardware) flags**
  - these are **sticky** → signal is maintained until explicitly reset by client

1. **overflow**: exact result of operation too large for model
2. **divide_by_zero**
3. **invalid**: operations like \( \infty \times 0, 0/0 \), whose result is a signaling NaN

4. **underflow**: result is finite, but too small to represent with full precision within model
   → store best result available
5. **inexact**: exact result cannot be represented without rounding
   → will be very common

**Exception handling**
- check exception flags and write code to deal appropriately with the situation
- halting execution – immediately or delayed (arbitrarily)
Fortran facilities – mapping the exception flags

- **Constants of opaque type** `ieee_flag_type`

  - `ieee_overflow`
  - `ieee_divide_by_zero`
  - `ieee_invalid`
  - `ieee_underflow`
  - `ieee_inexact`

  These constants need not be supported (always quiet).

- **Each flag can essentially have two states**
  - Signaling or non-signaling
Example: Division by zero

Without using IEEE facilities

subroutine invert(z)
  real, intent(inout) :: z
  z = 1.0 / z
end subroutine

xz = 0.0
write(*,'('''Inverting xz='', &
     E10.3)') xz
call invert(xz)
write(*,'('''Inverted value:'', &
     E10.3)') xz
stop

- exception may cause **halting**
  → control behaviour via compiler switch if possible

- STOP statement: if it is missing, it is implementation-dependent whether any exception status is reported at termination

exception may be triggered here

signaling flags are reported
Example cont'd: Division by zero

With use of IEEE facilities

subroutine invert(z)
    use, intrinsic :: &
    ieee_exceptions, only : &
    ieee_divide_by_zero, &
    ieee_set_flag, &
    ieee_get_flag
    real, intent(inout) :: z
    logical :: sig
    call ieee_set_flag( &
        ieee_divide_by_zero, .false.)
    z = 1.0 / z
    call ieee_get_flag( &
        ieee_divide_by_zero, sig)
    if (sig) write(*,*) &
        'FPE div by zero signaled'
    call ieee_set_flag( &
        ieee_divide_by_zero, .false.)
end subroutine

Invoking program

use, intrinsic :: ieee_features, &
    only : ieee_halting
use, intrinsic :: ieee_arithmetic

call ieee_set_halting_mode( &
    ieee_divide_by_zero, .false.)
if (.not. ieee_support_flag( &
    ieee_divide_by_zero, xz)) then
    stop 'FPE div by zero unsupp.'
end if
xz = 0.0
write(*,fmt='(''Invert xz='', &
    E10.3')') xz
call invert(xz)
write(*,fmt='(''Result is :'' , &
    E10.3')') xz

Output:

Invert xz= 0.000E+00
FPE div by zero signaled
Result is : Infinity
Handling of exceptions (1): Halting mode

```fortran
logical function ieee_support_halting(flag)

  type(ieee_flag_type), intent(in) :: flag
returns .true. if specified flag supports halting

subroutine ieee_get_halting_mode(flag, halting)

  type(ieee_flag_type), intent(in) :: flag
  logical, intent(out) :: halting
  halting is set to .true. if flag signaling causes halting

subroutine IEEE_set_halting_mode(flag, halting)

  type(ieee_flag_type) :: flag
  logical, intent(in) :: halting
  if halting is .true., the flag signaling will cause halting
  may only be called if ieee_support_halting(flag) is .true.
```

can be arrays
Handling of exceptions (2): Flag support

```fortran
logical function ieee_support_flag(flag [,x])
  type(ieee_flag_type), intent(in) :: flag
  returns .true. if specified flag supported [for kind of x]
end function ieee_support_flag

subroutine ieee_get_flag(flag, flag_value)
  type(ieee_flag_type), intent(in) :: flag
  logical, intent(out) :: flag_value
  flag_value is set to .true. if specified flag signals
end subroutine ieee_get_flag

subroutine ieee_set_flag(flag, flag_value)
  type(ieee_flag_type) :: flag
  logical, intent(in) :: flag_value
  if flag_value is .false., specified flag will be set to quiet
  will only set signaling if ieee_support_flag(flag) is .true.
end subroutine ieee_set_flag
```


Disambiguate two exceptions of the same kind

**Background:**
- Manipulating flags is an expensive operation.
- Trace exception for complete code blocks with many FP operations, and only a few potential failure points.
- More than one failure point with the same exception → need to record exceptions for disambiguation.

```plaintext
execute fast algorithm with potential FP exception
y = .../0.0
y_status(1) = ieee_set_flag()

ieee_get_status()

z = .../0.0
store to y_status(2)
```

```
second failure point
reset flag
```
Handling of exceptions (3):
Saving and restoring FP state

**Opaque derived type**
- `ieee_status_type`

**Object of that type stores**
- all exception flags
- rounding mode
- halting mode
  - complete FP state

**It cannot be directly accessed for information**

**Two transfer routines**

- **subroutine ieee_get_status( & status_value)**
  - `type(ieee_status_type), & intent(out) :: status_value`
  - reads FP state into status_value

- **subroutine ieee_set_status( & status_value)**
  - `type(ieee_status_type), & intent(in) :: status_value`
  - write FP state back to flags/registers
  - then, use `ieee_get_flag()` etc to retrieve information

```
ieee_set_status()
```

`y_status(1)` execute slower but safe algorithm

```
ieee_set_status()
```

`y_status(2)` unrecoverable error → abort

exec. sequence
Determine IEEE class

- Named constants of type ieee_class_type
  - ieee_signaling_nan
  - ieee_quiet_nan
  - ieee_negative_inf
  - ieee_negative_normal
  - ieee_negative_denormal
  - ieee_negative_zero
  - ieee_positive_zero
  - ieee_positive_denormal
  - ieee_positive_normal
  - ieee_positive_inf
  - ieee_other_value
  - unidentifiable (e.g., via I/O)

- This opaque type supports
  - assignment

- Elemental intrinsics:
  - type(ieee_class_type) &
    function ieee_class(x)
    - return the class a real number belongs to.

  - logical function ieee_is_
    finite
    nan
    negative
    normal
    (x)
  - identify FP class;
  - use e.g., to disambiguate multiple exceptions.
Further IEEE class functions

```
logical function &
ieee_unordered(x,y)

  returns .true. if one or both arguments are NaN
```

```
real( kind(x) ) function &
ieee_value(x, class)
real(...), intent(in) :: x

type(ieee_class_type), &
intent(in) :: class

  produce special number values (Infinity, NaN, denormals) if supported

  invoking the intrinsic should not trigger a FP signal

  values are processor-dependent, but are the same between different invocations with the same argument
```
Further IEEE intrinsics

- in part mirror-images of standard intrinsics
  
  nearest() → ieee_next_after()

- many elemental (i.e., applicable to arrays as well as scalars)

- please consult standard (section 14.11) for specification of these intrinsics

Example programs:

- see examples/day4/ieee

This is it for today!

tomorrow morning: Exercise session 8
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.:

```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

Type parameters are declared the same way as usual DT-components with the addition of specifying either the kind or len attribute

```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
```

- 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

module mod_matrix
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

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

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

disadvantage:
1 two type definitions
2 dynamic data only through allocatable or pointer attribute
Parameterized Derived Types:
Inquire Type parameters

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

```fortran
!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.:

```fortran
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

```fortran
!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

```fortran
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%c
          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 parameterized derived type with deferred len-parameter if they have the pointer or allocatable attribute.

```fortran
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
```

- 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.

```fortran
!type definition as in previous example
!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
```
Parameterized Derived Types: Default Type Parameters

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

```fortran
module pmatrixT
  private
  contains
    type, parameter (pmatrixT) :: k=real64, len=k=real64
    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 parameters 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
end module pmatrixT
```

- 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:
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
! usage, e.g.:  
type(mat_rT(real32,9,9,int64,80)), 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)','%m=',P%m)  
    write(*,'(counter)','%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).

tyep 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
```
Interoperation of Fortran with C
Overview of functionality defined in Fortran within C

<table>
<thead>
<tr>
<th>Area of semantics</th>
<th>within Fortran</th>
<th>within C</th>
</tr>
</thead>
<tbody>
<tr>
<td>function (procedure) call</td>
<td>invoke C function or interoperable Fortran procedure</td>
<td>invoke interoperable Fortran procedure</td>
</tr>
<tr>
<td>main program</td>
<td>only one: either Fortran or C</td>
<td></td>
</tr>
<tr>
<td>intrinsic data types</td>
<td>subset of Fortran types denoted as interoperable; not all C types are known</td>
<td>not all Fortran types may be known</td>
</tr>
<tr>
<td>derived data types</td>
<td>special attribute enforces interoperability with C struct types</td>
<td>„regular“ Fortran derived types not (directly) usable</td>
</tr>
<tr>
<td>global variables</td>
<td>access data declared with external linkage in C</td>
<td>access data stored in COMMON or module variable</td>
</tr>
<tr>
<td>dummy arguments</td>
<td>arrays or scalars</td>
<td>pointer parameters</td>
</tr>
<tr>
<td>dummy arguments</td>
<td>with VALUE attribute</td>
<td>non-pointer parameters</td>
</tr>
</tbody>
</table>
Overview continued

- Dealing with I/O:
  - Fortran record delimiters
  - STREAM I/O already dealt with

- Earlier attempts
  - F2C interface
  - `fortran.h` include file
  - proprietary directives

- C and Fortran pointers

<table>
<thead>
<tr>
<th>Semantics</th>
<th>within Fortran</th>
<th>within C</th>
</tr>
</thead>
<tbody>
<tr>
<td>C pointer</td>
<td>object of <code>type(c_ptr)</code></td>
<td><code>void *</code></td>
</tr>
<tr>
<td>C function pointer</td>
<td>object of <code>type(c_funptr)</code></td>
<td><code>void (*)()</code></td>
</tr>
</tbody>
</table>

- focus here is on: standard conforming Fortran/C interoperation

- module functions are provided via an intrinsic module to map data stored inside these objects to Fortran POINTERs and procedure pointers
The concept of a companion processor

- **Used for implementing C interoperable types, objects and functions**
  - it must be possible to describe function interfaces via a C prototype

- **Companion may be**
  - a C processor
  - another **Fortran processor** supporting C interoperation
  - or **some other** language supporting C interoperation

- **Note:**
  - different C processors may have different ABIs and/or calling conventions
  - therefore not all C processors available on a platform may be suitable for interoperation with a given Fortran processor
C-Interoperable intrinsic types

Example program:

```fortran
program myprog
  use, intrinsic :: iso_c_binding
  integer(c_int) :: ic
  real(c_float) :: rc4
  real(c_double), allocatable :: a(:)
  character(c_char) :: cc
  allocate(a(ic), …)
  call my_c_subr(ic,a)
end program
```

- a module provided by the Fortran processor
- further stuff omitted here – will be shown later
- might be implemented in Fortran or C. Will show a C implementation later
## Mapping of some commonly used intrinsic types

### via KIND parameters
- integer constants defined in ISO_C_BINDING intrinsic module

<table>
<thead>
<tr>
<th>C type</th>
<th>Fortran declaration</th>
<th>C type</th>
<th>Fortran declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integer(c_int)</td>
<td>char</td>
<td>character(len=1,kind=c_char)</td>
</tr>
<tr>
<td>long int</td>
<td>integer(c_long)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>integer(c_size_t)</td>
<td>_Bool</td>
<td>logical(c_bool)</td>
</tr>
<tr>
<td>[un]signed char</td>
<td>integer(c_signed_char)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>real(c_float)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>real(c_double)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- a **negative** value for a constant causes compilation failure (e.g., because no matching C type exists, or it is not supported)
- a standard-conforming processor must only support `c_int`
- compatible C types derived via typedef also interoperate
Calling C subprograms from Fortran:

a simple interoperable interface

Assume a C prototype

```c
void My_C_Subr(int, double []);
```

Need a Fortran interface

- explicit interface
- `BIND(C, name='...')` attribute
  - suppress Fortran name mangling
  - label allows mixed case name resolution and/or renaming
    (no label specified → lowercase Fortran name is used)
  - cannot have two entities with the same binding label

VALUE attribute/statement

- create copy of argument
- some limitations apply
  (e.g., cannot be a POINTER)

Scalar vs. array pointers

- no unique interpretation in C
- check API documentation
Functions vs. subroutines and compilation issues

- **C function with `void` result**
  - may interoperate with a Fortran subroutine

- **All other C functions**
  - may interoperate with a Fortran function

**Link time considerations**

- **recommendation:** perform linkage with Fortran compiler driver to assure Fortran RTL is linked in

- may need a special compiler link-time option if main program is in C (this is processor-dependent)
Passing arrays between Fortran and C (1)

Return to previous example:

- assume that six array elements have been allocated
- remember layout in memory: **contiguous** storage sequence

Inside C

- formal parameter `double d[]` uses zero-based indexing
  (C ignores **any** lower bound specification in the Fortran interface!)

Note:

- in a call from Fortran, a non-contiguous array (e.g. a section) may be used → will be automatically compactified (copy-in/out)
- need to do this manually in calls from C
Multidimensional arrays

Example Fortran interface

```fortran
interface
  subroutine solve_mat( &
    b, n1, n2) bind(c)
  use, intrinsic :: iso_c_binding
  integer(c_int), value :: n1, n2
  real(c_double) :: b(n1,*)
end subroutine solve_mat
end interface
```

Two possible C prototypes

```c
void solve_mat(double *, \int, int);

void solve_mat(double [][*], \int, int);
```

C99 VLA (variable length array)

Assume actual argument in call from Fortran:

```fortran
double precision :: rm(0:1,3)
: call solve_mat(rm, 2, 3)
```

```
0 1 3 5
1 2 4 6
```

"column major" mapping of array indices to storage index

```
1 2 3
```

2nd index

```
0 1 3 5
```

1st index
First alternative – manual mapping

example implementation:

```c
void solve_mat(double *d, int n1, int n2) {
    double **dmap;
    int i, k;
    dmap = (double **) malloc(n2 * sizeof(double *));
    for (i=0; i<n2; i++) {
        dmap[i] = d + n1 * i;
    // now access array elements via dmap
    for (k=0; k<n1; k++) {
        dmap[i][k] = ...;
    }
    }
    free (dmap);
}
```

LHS is of type double

force **dmap to contiguous storage layout

1st index

"row major" mapping of array indices to storage index

© 2009-19 LRZ

Advanced Fortran Topics - LRZ section

215
Accessing multidimensional array data in C

- **Second alternative – C99 VLA**
  - example implementation:

```c
void solve_mat(double d[][n1], int n1, int n2) {
    int i, k;
    // directly access array elements
    for (i=0; i<n2; i++) {
        for (k=0; k<n1; k++) {
            d[i][k] = …;
        }
    }
    ...
}
```

- **Caveat for use of ** (pointer-to-pointer):**
  - **in general** this describes a non-contiguous storage sequence → **cannot**
    be used as interoperable array parameter

  ```c
dmap[i] = (double *) malloc(...);
```

© 2009-19 LRZ

Advanced Fortran Topics - LRZ section
Handling of strings (1)

- Remember: character length must be 1 for interoperability

- Example: C prototype

```c
int atoi(const char *);
```

- matching Fortran interface declares c_char entity an assumed size array

```fortran
interface
  integer(c_int) function atoi(in) bind(c)
    use, intrinsic :: iso_c_binding
    character(c_char), dimension(*) :: in
  end function
end interface
```
Handling of strings (2)

- Invoked by

```fortran
use, intrinsic :: iso_c_binding
character(len=::,kind=c_char), allocatable :: digits
allocate(character(len=5) :: digits)
digits = c_char_1234 // c_null_char
i = atoi(digits) ! i gets set to 1234
```

- **special exception** (makes use of storage association): actual argument may be a scalar character string

- Character constants in ISO_C_BINDING with C-specific meanings

<table>
<thead>
<tr>
<th>Name</th>
<th>Value in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_null_char</td>
<td>‘\0’</td>
</tr>
<tr>
<td>c_new_line</td>
<td>‘\n’</td>
</tr>
<tr>
<td>c_carriage_return</td>
<td>‘\r’</td>
</tr>
</tbody>
</table>

C string needs terminator

most relevant subset
C Interoperation with derived types

**Example:**

```fortran
use iso_c_binding :
type, bind(c) :: dtype
  integer(c_int) :: ic
  real(c_double) :: d(10)
end type dtype

is interoperable with

typedef struct {
  int i;
  double dd[10];
} dtype_c;

and typed variables can be used
e.g., in argument lists
```

**Notes:**

- naming of types and components is irrelevant
- `bind(c)` cannot have a label in this context. It cannot be specified together with `sequence`
- position of components must be the same
- type components must be of interoperable type
Interoperation with derived types: Restrictions

- In this context, Fortran type components must **not be**
  - pointers or allocatable
  - zero-sized arrays
  - type bound procedures

- **Fortran type must not be**
  - extension of another type (and an interoperable type cannot itself be extended!)

- **C types which cannot interoperate:**
  - union types
  - structs with bit field components
  - structs with a flexible array member
Handling non-interoperable data – the question now is ...

- when and how to make objects of the (non-interoperable!) Fortran type

```fortran
  type :: fdyn
    real(c_float), allocatable :: f(:)
  end type fdyn
```

available within C

- when and how to make objects of the analogous C type

```c
  typedef struct cdyn {
    int len;
    float *f;
  } Cdyn;
```

available within Fortran
Case 1: Data only accessed within C

API calls are

```c
Cdyn *Cdyn_create(int len) {
    this = (Cdyn *) malloc(...);
    this->f = (float*) malloc(...);
    return this;
}
void Cdyn_add(Cdyn *v, ...) {
    v->f[i] = ...;
}
```

Assumptions:

- want to call from Fortran
- but no access to type components needed within Fortran

Required Fortran interface

```fortran
use, intrinsic :: iso_c_binding
interface
    type(c_ptr) function &
    cdyn_create(len) bind(c,...)
    import :: c_int, c_ptr
    integer(c_int), value :: len
end function
subroutine cdyn_add(h, ...) &
    bind(c,...)
    import :: c_ptr
    type(c_ptr), value :: h
end subroutine
end interface
```

Object of type `c_ptr` requires value attribute here.
Typeless C pointers in Fortran

Opaque derived types defined in ISO_C_BINDING:
- c_ptr: interoperates with a void * C object pointer
- c_funptr: interoperates with a C function pointer.

Useful named constants:
- c_null_ptr: C null pointer
- c_null_funptr: C null function pointer

Logical module function that checks pointer association:
- c Associated(c1[,c2])
  - value is .false. if c1 is a C null pointer or if c2 is present and points to a different address. Otherwise, .true. is returned

Typical usage:

```fortran
type(c_ptr) :: res
res = get_my_ptr( ... )
if (c_associated(res)) then
  ! do work with res
else
  stop 'NULL pointer produced by get_my_ptr'
end if
```
Case 1 (cont'd): Client usage

```fortran
use, intrinsic :: iso_c_binding
:
type(c_ptr) :: handle
:
handle = cdyn_create(5_c_int)
if (c_associated(handle)) then
    call cdyn_add(handle,...)
end if
call cdyn_destroy(handle)
```

- **Typeless „handle“ object**
  - because objects of (nearly) any type can be referenced via a `void *`, no matching type declaration is needed in Fortran

- **Design problem:**
  - no disambiguation between different C types is possible → loss of type safety

all memory management done in C

see exercise for a possible solution
Module ISO_C_BINDING provides module procedures

- Setting up a mapping between a Fortran object and a C pointer

Module ISO_C_BINDING provides module procedures

- pointer association (blue arrow) is set up as a result of their invocation (green arrows)

Fortran object (f)

- must be a TARGET
- same type and type parameters
- lower bounds are 1 if an array pointer

Fortran POINTER (fptr)

- same type and type parameters
- lower bounds are 1 if an array pointer

C pointer (cp)

- type(c_ptr) :: cp
  cp = c_loc(f)

- c_loc() produces C address of F

(A) for scalar Fortran object

call c_f_pointer(cp,fptr)

(B) for array Fortran object

call c_f_pointer(cp,fptr,shape)

information must be separately provided (integer array)
Two scenarios are covered by `c_loc/c_f_pointer`

1. **Fortran object is of interoperable type and type parameters:**
   - variable with `target` attribute,
   - or allocated variable with `target` attribute, non-zero length,
   - or associated scalar pointer

2. **Fortran object is a non-interoperable, non-polymorphic scalar without length type parameters:**
   - non-allocatable, non-pointer variable with `target` attribute,
   - or an allocated allocatable variable with `target` attribute,
   - or an associated pointer.

In scenario 1, the object might also have been created within C (Fortran target then is anonymous). In any case, the data can be accessed from C.

Nothing can be done with such an object within C.
Case 2: Data accessed only within Fortran

Fortran Library definition

```fortran
module mylib
  use, intrinsic :: &
    iso_c_binding
  type :: fdyn
    real, allocatable :: f(:)
  end type fdyn
contains
  ! continued to the right
  type (c_ptr) function &
    fdyn_create(len) bind(c, ...)
  integer(c_int), value :: len
  type(fdyn), pointer :: p
  allocate(p)
  allocate(p%f(len))
  fdyn_create = c_loc(p)
end function
end module mylib
```

- noninteroperable derived data type
- provide an interoperable constructor written in Fortran

- Pointer goes out of scope
  - but target remains reachable via function result

C prototype:

```c
void *fnew_stuff(int);
```
Case 2 (cont'd): Retrieving the data

Client code in C:

```c
void *fhandle;
int len = 5;

fhandle = Fdyn_create(len);
Fdyn_print(fhandle);
```

- can have multiple handles to different objects at the same time (thread-safeness)
- again no matching type needed on client
- require Fortran implementation of `Fdyn_print()`

... here it is:

```fortran
subroutine fdyn_print(h) bind(c,...)
  type(c_ptr), value :: h
  type(fdyn), pointer :: p

  call c_f_pointer(h, p)
  if (allocated(p%f)) then
    write(*,fmt=...) p%f
  end if
end subroutine
```

... and must not forget to

- implement „destructor“ (in Fortran)
- and call it (from C or Fortran) for each created object
- to prevent memory leak
Warning on inappropriate use of `c_loc()` and `c_f_pointer()`

- **With these functions,**
  - it is possible to subvert the type system (don't do this!) (push in object of one type, and extract an object of different type)
  - it is possible to subvert rank consistency (don't do this!) (push in array of some rank, and generate a pointer of different rank)

- **Implications:**
  - implementation-dependent behaviour
  - security risks in executable code

- **Recommendations:**
  - use with care (testing!)
  - encapsulate use to well-localized code
  - don't expose use to clients if avoidable
Case 3: Accessing C-allocated data in Fortran

- We haven't gone the whole way towards fully solving the problem
  - won't actually do so in this talk

- Return to Case 1:

```
typedef struct Cdyn {
   int len;
   float *f;
} Cdyn;
```

```
void Cdyn_print(Cdyn *);
```

- and implement the function with above C prototype in Fortran
- → need read and/or write access to data allocated within the C-defined structure
- allocation is performed as described in Case 1
Case 3 (cont'd): Fortran implementation

**Required type definition:**

```fortran
type, bind(c) :: cdyn
  integer(c_int) :: len
  type(c_ptr) :: f
end type cdyn
```

**Notes:**

- Note the `intent(in)` for this (refers to association of `c_ptr`; the referenced data can be modified)
- **scenario 1** applies for `c_f_pointer` usage

**Implementation:**

```fortran
subroutine cdyn_print(this) bind(c,name='Cdyn_print')
  type(cdyn), intent(in) :: this
  real(c_float), pointer :: cf(:)
  ! associate array pointer cf with this%f
  call c_f_pointer( this%f, cf, [this%len] )
  ! now do work with data pointed at by this%f
  write(*,fmt=...) cf
end subroutine cdyn_print
```

following now: Exercise session 10
Procedure arguments and pointers (1)

- **Procedure argument: a function pointer in C**
  - could have a fixed or variable interface
  - example C prototype:
    
    ```c
    double integrate(double, double, void *,
                     double (*)(double, void *));
    ```

- **Matched by interoperable Fortran interface**

  ```fortran
  real(c_double) function integrate(a, b, par, fptr) bind(c)
    real(c_double), value :: a, b
    type(c_ptr), value :: par
    type(c_funptr), value :: fptr
  end function
  ```

- **Note:**
  - an interface with a Fortran procedure dummy argument is **not** interoperable (even if the dummy procedure has the BIND(C) attribute)
Module ISO_C_BINDING provides module procedures

- input for `c_funloc` must be an **interoperable** Fortran procedure; can also be an associated procedure pointer
- pointer association (blue arrow) is set up as a result of their invocation (green arrows)
Assuming the function interface

```fortran
real(c_double) function f(x, par) bind(c)
    real(c_double), value :: x
    type(c_ptr), value :: par
end function
```

the invocation reads

```fortran
type(c_funptr) :: fp
fp = c_funloc(f)
res = integrate( a, b, par, fp )
```

or, more concisely

```fortran
res = integrate( a, b, par, c_funloc(f) )
```
Another procedure pointer example

- **C function pointer used as type component**

```c
typedef struct {
    double (*f)(double, void *);
    void *par;
} ParFun;
```

- **Matching type definition in Fortran:**
  - requires use of component of type `c_funptr`

```fortran
type, BIND(C) :: parfun
    type(c_funptr) :: f
    type(c_ptr) :: par
end type
```
Invoking the C-associated procedure from Fortran

Example:

```fortran
type(parfun) :: o_pf
o_pf%f = c_funloc(my_function)
o_pf%par = c_loc(…)
```

where `my_function` should have the **same** interface in Fortran and C, respectively.

```fortran
ParFun *o_pf;
o_pf->f = my_function;
o_pf->par = (void *) …;
```

```fortran
type(parfun) :: o_pf
type(c_ptr) :: par
procedure(my_function), pointer :: pf

: ! initialize o_pf, par within C or Fortran
call c_f_procpointer(o_pf%f, pf)
y = pf(2.0_dk, par)
```
Interoperation of global data (1): COMMON blocks

- **Defining C code:**

  ```c
  struct coord{
    float xx, yy
  };
  struct coord csh;
  ```

  - do not place in include file
  - reference with `external` in other C source files

- **Mapping Fortran code**

  ```fortran
  real(c_float) :: x, y
  common /csh/ x, y
  bind(c) :: /csh/
  ```

  - **BIND statement** (possibly with a label) resolves to the same linker symbol as defined in C → **same memory address**
  - memory layout may be different as for „traditional“ sequence association
Interoperation of global data (2): Module variables

- **Defining C code:**

  ```c
  int ic;
  float Rpar[4];
  ```

- **Mapping Fortran code:**

  ```fortran
  module modGlobals
    use, intrinsic :: iso_c_binding
    integer(c_int), bind(c) :: ic
    real(c_float) :: rpar(4)
    bind(c, name='Rpar') :: rpar
  end module
  ```

- Global binding can be applied to objects of interoperable type and type parameters. Variables with the ALLOCATABLE/POINTER attribute are not permitted in this context.
Enumeration

Set of integer constants

- only for interoperation with C

```fortran
enum, bind(c)
  enumerator :: red=4, blue=9
  enumerator :: yellow
end enum
```

- integer of same kind as used in C enum
- value of yellow is 10
- not hugely useful
Extension of interoperability with C in


**Motivations:**
- enable a standard-conforming MPI (3.1) Fortran interface
- permit C programmers (limited) access to „complex“ Fortran objects

<table>
<thead>
<tr>
<th>Area of semantics</th>
<th>within Fortran</th>
<th>within C</th>
</tr>
</thead>
<tbody>
<tr>
<td>dummy argument</td>
<td>assumed shape/length or deferred shape/length</td>
<td>pointer to a <strong>descriptor</strong></td>
</tr>
<tr>
<td></td>
<td><strong>POINTER or ALLOCATABLE</strong></td>
<td></td>
</tr>
<tr>
<td>dummy argument</td>
<td>assumed rank</td>
<td>pointer to a <strong>descriptor</strong></td>
</tr>
<tr>
<td>dummy argument</td>
<td>assumed type</td>
<td>either <code>void *</code> or pointer to a <strong>descriptor</strong></td>
</tr>
<tr>
<td>dummy argument</td>
<td><strong>OPTIONAL attribute</strong> no <strong>VALUE attribute permitted</strong></td>
<td>use a NULL actual or check formal for being NULL</td>
</tr>
<tr>
<td>dummy argument of type c_ptr or c_funptr</td>
<td>non-interoperable data or procedure</td>
<td>handle only, no access to data or procedure</td>
</tr>
<tr>
<td>non-blocking procedures</td>
<td><strong>ASYNCHRONOUS attribute</strong></td>
<td>not applicable</td>
</tr>
</tbody>
</table>
Accessing Fortran infrastructure from C: the source file ISO_Fortran_binding.h

Example Fortran interface

```fortran
subroutine process_array(a) BIND(C)
  real(c_float) :: a(:,:)
end subroutine
```

Matching C prototype

```c
#include <ISO_Fortran_binding.h>

void process_array(CFI_cdesc_t *a);
```

Implementation of procedure might be in C or in Fortran

For an implementation in C, the header provides access to

- type definition of descriptor (details upcoming …)
- macros for type codes, error states etc.
- prototypes of library functions that generate or manipulate descriptors

Reserved namespace: CFI_

Within a single C source file,

- binding is only possible to one given Fortran processor (no binary compatibility!)
Members of the C descriptor

- Exposes internal structure of Fortran objects
  - not meant for tampering → please use API

```
void *base_addr
size_t elem_len
int version
CFI_rank_t rank
CFI_type_t type
CFI_attribute_t attribute
CFI_dim_t dim[]
```

- `void *base_addr`: start address of object's data memory area
- `size_t elem_len`: size of a scalar in bytes
- `int version`: implementation can check object against CFI_VERSION
- `CFI_rank_t rank`: zero for a scalar, otherwise rank of the array
- `CFI_type_t type`: zero, except maybe for deferred-shape objects
- `CFI_attribute_t attribute`: stride multiplier: distance between array elements in specified dimension (in units of bytes)
- `CFI_dim_t dim[]`: lattice structure of array (of size ≥ “rank“)
- `CFI_index_t lower_bound`: zero, except maybe for deferred-shape objects
- `CFI_index_t extent`: stride multiplier: distance between array elements in specified dimension (in units of bytes)
Type and attribute members

Type code macros
- most commonly used:

<table>
<thead>
<tr>
<th>Name</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI_type_int</td>
<td>int</td>
</tr>
<tr>
<td>CFI_type_long</td>
<td>long int</td>
</tr>
<tr>
<td>CFI_type_size_t</td>
<td>size_t</td>
</tr>
<tr>
<td>CFI_type_float</td>
<td>float</td>
</tr>
<tr>
<td>CFI_type_double</td>
<td>double</td>
</tr>
<tr>
<td>CFI_type_Bool</td>
<td>_Bool</td>
</tr>
<tr>
<td>CFI_type_char</td>
<td>char</td>
</tr>
<tr>
<td>CFI_type_cptr</td>
<td>void *</td>
</tr>
<tr>
<td>CFI_type_struct</td>
<td>Interoperable C structure</td>
</tr>
<tr>
<td>CFI_type_other (&lt;0)</td>
<td>Not otherwise specified</td>
</tr>
</tbody>
</table>

Attribute of dummy object

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI_attribute_allocatable</td>
</tr>
<tr>
<td>CFI_attribute_pointer</td>
</tr>
<tr>
<td>CFI_attribute_other</td>
</tr>
</tbody>
</table>

Beware: attribute value of actual must match up exactly with that of dummy (different from Fortran) → may need to create descriptor copies

© 2009-19 LRZ Advanced Fortran Topics - LRZ section 243
Using the descriptor to process array elements (1)

**Fortran reference loop within** `process_array()`:

```fortran
do k=1, ubound(a, 2)
   do i=1, ubound(a, 1)
      ... = a(i, k) * ...
   end do
end do
```

**C implementation variant 1:**

```c
for (k = 0; k < a->dim[1].extent; k++) {
   for (i = 0; i < a->dim[0].extent; i++) {
      CFI_index_t subscripts[2] = { i, k };
      ... = *((float *) CFI_address( a, subscripts )) * ...;
   }
}
```

- `CFI_address()` returns `(void *)` address of array element indexed by specified (valid!) subscripts
- `dim[].lower_bound` will be needed for pointer/allocatable objects
C implementation variant 2:

- start out from beginning of array

```c
char *a_ptr = (char *) a->base_addr;
```

and use pointer arithmetic to process it:

```c
char *a_aux;
for (k = 0; k < a->dim[1].extent; k++) {
    a_aux = a_ptr;
    for (i = 0; i < a->dim[0].extent; i++) {
        ... = *((float *) a_ptr) * ...;
        a_ptr += a->dim[0].sm;
    }
    a_ptr = a_aux + a->dim[1].sm;
}
```

- non-contiguous arrays require use of stride multipliers (next slide illustrates why)

- stride multipliers in general may not be an integer multiple of the element size \( \rightarrow \) always process in units of bytes
Memory layouts for assumed shape objects

- **Actual argument is a complete array (0:1,3)**
  - *base_addr
  - dim[0] 1
  - sm=8B
  - dim[1]
  - 0 1 2
  - indicates array element sequence of dummy argument
  - CFI_is_contiguous(a) returns 1

- **Actual argument is an array section (0::2,1::3) of (0:2,9)**
  - *base_addr
  - dim[0] 1
  - sm=24B
  - dim[1]
  - 0 1 2
  - all “orange“ storage units are not part of the object, but are exposed by descriptor → do not touch!
  - CFI_is_contiguous(a) returns 0
Creating a Fortran object within C

May be necessary to invoke a Fortran procedure from C

Step 1: create a descriptor

CFI_CDESC_T(2) A;

CFI_cdesc_t *a = (CFI_cdesc_t *) &A;

Step 2: establish object’s properties

• prototype of function to be used for this is

```c
int CFI_establish(
    CFI_cdesc_t *dv,
    void *base_addr,
    CFI_attribute_t attribute,
    CFI_type_t type,
    size_t elem_len,
    CFI_rank_t rank,
    const CFI_index_t extents[]
);
```

• many usage patterns
• if fully defined, result is always a contiguous object
• function result is an error indicator (CFI_SUCCESS → OK)
## Table of return value macros

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Explanation of error</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CFI_SUCCESS</code></td>
<td>No error detected.</td>
</tr>
<tr>
<td><code>CFI_ERROR_BASE_ADDR_NULL</code></td>
<td>The base address member of a C descriptor is a null pointer in a context that requires a non-null pointer value.</td>
</tr>
<tr>
<td><code>CFI_ERROR_BASE_ADDR_NOT_NULL</code></td>
<td>The base address member of a C descriptor is not a null pointer in a context that requires a null pointer value.</td>
</tr>
<tr>
<td><code>CFI_INVALID_ELEM_LEN</code></td>
<td>The value supplied for the element length member of a C descriptor is not valid.</td>
</tr>
<tr>
<td><code>CFI_INVALID_RANK</code></td>
<td>The value supplied for the rank member of a C descriptor is not valid.</td>
</tr>
<tr>
<td><code>CFI_INVALID_TYPE</code></td>
<td>The value supplied for the type member of a C descriptor is not valid.</td>
</tr>
<tr>
<td><code>CFI_INVALID_ATTRIBUTE</code></td>
<td>The value supplied for the attribute member of a C descriptor is not valid.</td>
</tr>
<tr>
<td><code>CFI_INVALID_EXTENT</code></td>
<td>The value supplied for the extent member of a <code>CFI_dim_t</code> structure is not valid.</td>
</tr>
<tr>
<td><code>CFI_INVALID_DESCRIPTOR</code></td>
<td>A general error condition for C descriptors.</td>
</tr>
<tr>
<td><code>CFI_ERROR_MEM_ALLOCATION</code></td>
<td>Memory allocation failed.</td>
</tr>
<tr>
<td><code>CFI_ERROR_OUT_OF_BOUNDS</code></td>
<td>A reference is out of bounds.</td>
</tr>
</tbody>
</table>
Example: create rank 2 assumed shape array

```c
#define DIM1 56
#define DIM2 123

CFI_CDESC_T(2) A; /*!< 2 is the minimum value needed */
CFI_cdesc_t *a = (CFI_cdesc_t *) &A;
CFI_index_t extents[2] = { DIM1, DIM2 }; /*!< shape of rank 2 array */

float *a_ptr = (float *) malloc(DIM1*DIM2*sizeof(float)); /*!< heap allocation within C */
...
/*! initialize values of *a_ptr */

CFI_establish( a, (void *) a_ptr,
CFI_attribute_other,
CFI_type_float,
0, /*!< elem_len is ignored here */
2, /*!< rank as declared in Fortran */
extents );
/*! have a fully defined object now */

process_array(a);

free(a_ptr); /*!< object becomes invalid */
```
Allocatable objects

- Typically only needed if Fortran API defines a „factory“:

```fortran
type, bind(c) :: qbody
  real(c_float) :: mass
  real(c_float) :: position(3)
end type

interface
  subroutine qbody_factory(this, fname) bind(c)
    type(qbody), allocatable, intent(out) :: this(:,:)
    character(c_char, len=*) , intent(in) :: fname
  end subroutine
end interface
```

- matching C prototype:

```c
void qbody_factory(CFI_cdesc_t *, CFI_cdesc_t *)
```

typedef struct {
  float mass;
  float position[3];
} qbody;
Example: create an allocatable entity and populate it

```fortran
char fname_ptr[] = "InFrontOfMyHouse.dat";

CFI_cdesc_t *pavement =
    (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
CFI_cdesc_t *fname =
    (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(0)));

CFI_establish(pavement, NULL, CFI_attribute_allocatable,
              CFI_type_struct, sizeof(qbody), /* derived type object size */
              2, NULL);

CFI_establish(fname, fname_ptr, CFI_attribute_other,
              CFI_type_char, strlen(fname_ptr),    /* a char has one byte */
              0, NULL);

qbody_factory (pavement, fname); /* object is created */
...

CFI_deallocate(pavement);       /* process pavement */

free(pavement); free(fname);
```

must start out unallocated

shape is deferred

no auto-deallocation of objects allocated in C
An implementation of `qbody_factory()` in C

```c
void qbody_factory(CFI_cdesc_t *this, CFI_cdesc_t *fname_str) {
    char *fname = (char *) fname_str->base_addr;
    CFI_index_t lowerbds[2], upperbds[2];
    ... /* open file *fname and read in bounds information */
    if (this->base_addr != NULL) CFI_deallocate(this);
    /* emulate INTENT(OUT) semantics for C-to-C calls */
    CFI_allocate(this, lowerbds, upperbds, 0);
    /* object is now allocated */
    ... /* read object data from file *fname */
}
```

Feasible because of supplied function `CFI_allocate()`:

- last argument is an element length, which is ignored unless the type member is `CFI_type_char`. In the latter case, its value becomes the element length of the allocated deferred-length (!) string.

following now: Exercise session 11
Handling Fortran POINTERS within C

- **Anonymous target**
  - create descriptor with `CFI_attribute_pointer`, then apply `CFI_allocate()`/`CFI_deallocate()`

- **Point at an existing target**
  ```fortran
  real(c_float), TARGET :: t(:)
  real(c_float), POINTER :: p(:)
  p(3:) => t
  ```
  ```c
  CFI_index_t lower_bounds[1] = { 3 };
  status = CFI_setpointer( p, t, lower_bounds );
  ```

- **t** must describe a fully valid object
- **p** must be an established descriptor with `CFI_attribute_pointer` and for the same type as t.

**Beware:** No compile-time type safety is provided.
Certain inconsistencies may be diagnosed at run time
→ check return value of `CFI_setpointer()`
Assumption:

- `arr` describes an assumed-shape rank 3 array

Create a descriptor for the section `arr(3:,4,:,:2)`

```c
CFI_cdesc_t *section =
    (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
CFI_index_t lower_bounds[3] = { 2, 3, 0 };
CFI_index_t upper_bounds[3] =
    { arr->dim[0].extent - 1, 3, arr->dim[2].extent - 1 };
CFI_index_t strides[3] = { 1, 0, 2 };

CFI_establish( section, NULL, CFI_attribute_other,
               arr->type, arr->elem_len, 2, NULL );
/* section here is an undefined object */
CFI_section( section, arr, lower_bounds, upper_bounds, strides );
/* now, section is defined */
```

Zero stride indicates a subscript. For this dimension, lower and upper bounds must be equal.
Creating subobjects in C (2)

**Type component selection**
- `pavement(:,1)` from the type(`qbody`) object `pavement`
- A rank-2 array of intrinsic type `real(c_float)`

```c
CFI_cdesc_t *pos_1 =
  (CFI_cdesc_t *) malloc(sizeof(CFI_CDESC_T(2)));
size_t elem_len = 0;
CFI_establish(pos_1, NULL, CFI_attribute_other,
              CFI_type_float, elem_len, 2, NULL);
/* pos_1 here is an undefined object */

size_t displacement = offsetof(qbody, position[0]);
CFI_select_part(pos_1, pavement, displacement, elem_len);
/* now, pos_1 is defined */
```

Displacement:
- `sm=16B`
- `pavement(1:4,1:1)`

© 2009-19 LRZ
Enables invocation of appropriately declared object

```fortran
subroutine process_allranks(ar, ...)  
   real :: ar(..)
   ...
   write(*,*) RANK(ar)
end subroutine
```

ar cannot (currently) be referenced or defined within Fortran. However, some intrinsics can be invoked.

with arrays of any rank, or even a scalar:

```fortran
real :: xs, x1(4), x2(ndim, 4)
call process_allranks(xs, ...)  
call process_allranks(x1, ...)  
call process_allranks(x2, ...)
```

avoid need for writing many specifics for a named interface
What arrives on the C side?

Assuming the procedure interface is made BIND(C):

- descriptor always contains well-defined rank information

```c
void process_allranks(CFI_cdesc_t *ar, ...) {
    switch( ar->rank )
    case 1:
        ... /* process single loop nest */
    case 2:
        ... /* process two nested loops */
    default:
        printf(“unsupported rank value\n”);
        exit(1);
}
```

- deep loop nests can be avoided for contiguous objects, but the latter is not assured

C implementation currently the only way to get at the data ...
Assumed size actual argument

- **Special case:**
  - size of (contiguous) assumed-size object is not known

  ```fortran
  real :: x2(ndim, *)
  call process_allranks(x2, ..., ntot)
  ```

- **A descriptor with following properties is constructed:**
  - `SIZE(ar,DIM=RANK(ar))` has the value `-1`
  - `UBOUND(ar,DIM=RANK(ar))` has the value `UBOUND(ar,DIM=RANK(ar)) - 2`
Assumed type dummy arguments

**Declaration with TYPE(*)**
- an unlimited polymorphic object → actual argument may be of any type
- dynamic type cannot change → no POINTER or ALLOCATABLE attribute is permitted

**Corresponding object in interoperating C call:**
- two variants are possible

```
TYPE(*), DIMENSION(*) :: obj
TYPE(*), DIMENSION(:) :: obj
TYPE(*) :: obj
TYPE(*), DIMENSION(..) :: obj
```

```
void *obj
CFI_cdesc_t *obj
```

or rank 2, 3, ...
Example: direct interoperation with MPI_Send

C prototype as specified in the MPI standard

```c
int MPI_Send( const void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm );
```

Matching Fortran interface:

```fortran
INTEGER(c_int) FUNCTION C_MPI_Send( buf, count, datatype, dest, &
                                      tag, comm ) BIND(C, name="MPI_Send")
                          TYPE(*), DIMENSION(*) , INTENT(IN) :: buf
                          INTEGER(c_int), VALUE :: count, dest, tag
                          TYPE(MPI_Datatype), VALUE :: datatype
                          TYPE(MPI_Comm), VALUE :: comm
END FUNCTION C_MPI_Send
```

- assumes interoperable types MPI_Datatype etc.
- actual argument may be array or scalar
- non-contiguous actuals are compactified

array temps are a problem for non-blocking calls

size of memory area specified by count and datatype
MPI-3 Fortran interface for MPI_send

```
SUBROUTINE MPI_Send( buf, count, datatype, dest, & 
tag, comm, ierror )
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN):: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
END FUNCTION MPI_Send
```

- **Invocation of MPI_Send**
  - now possible also with **array section** actual arguments without need for copy-in/out

- **Could add BIND(C) to the interface for a C implementation**
  - assuming int matches default Fortran integer
  - the MPI standard doesn‘t do this, though
void mpi_send( CFI_cdesc_t *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, int *ierror ) {
    int ierror_local;
    MPI_Datatype disc_type;
    if ( CFI_is_contiguous( buf ) ) {
        ierror_local = MPI_Send( buf->base_addr, count, datatype, dest, tag, comm );
    } else {
        ... /* use descriptor information to construct disc_type from datatype (e.g. via MPI_Type_create_subarray) */

        ierror_local = MPI_Send( buf->base_addr, count, disc_type, dest, tag, comm );
        ... /* clean up disc_type */
    }
    if (ierror != NULL) *ierror = ierror_local;
}
Automatized translation of C include files to Fortran interface modules

- Requires a specialized tool
  - for example, Garnet Liu’s LLVM-based tool, see [https://github.com/Kaiveria/h2m-Autofortran-Tool](https://github.com/Kaiveria/h2m-Autofortran-Tool)

- C include files can have stuff inside that is not covered by interoperability
  - only a subset can be translated

- Topic goes beyond the scope of this course
Final remarks

- **Interoperation with C++**
  - no direct interoperation with C++-specific features is possible
  - you need to write C-like bridge code
  - declare C-style functions „extern C“ in your C++ sources
  - explicit linkage of C++ libraries will be needed if the Fortran compiler driver is used for linking

- **Vararg interfaces**
  - are not interoperable with any Fortran interface
  - you need to write glue code in C
Shared Libraries and Plug-ins
What is a shared library?

- Executable code in library
  - is shared between all programs linked against the library (instead of residing in the executable)
  - this does not apply to data entities

- Advantages:
  - save memory space
  - save on access latency
  - bug fixes in library code do not require relinking the application

- Disadvantages:
  - higher complexity in handling the build and packaging of applications
  - (need to distribute shared libraries together with the linked application)
  - not supported (in analogous manner) on all operating environments
  - (will focus on ELF-based Linux in this talk)
  - special compilation procedure is required for library code

ELF → executable and linkable format
Compatibility issues

- **Causes of incompatibility**
  - change function interface
  - remove function interface
  - (adding a new function is no problem)
  - change in type definitions or data items
  - (exception: storage association is preserved)
  - changes in function behaviour

- **If one of these happens,**
  - a mechanism is available to indicate the library is not compatible

- **Concept of soname**
  - naming scheme for shared libraries
    - lib<name>.so.X.Y.Z
      - major version
      - minor version
      - release number
  - soname will typically be
    - lib<name>.so.X
  - and the latest version of the library with the same soname should be picked up at linkage
Example (1): building a library

- Assume you have
  - a source file `mylib.f90`

 Implementation-dependent options

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compilation</th>
<th>Linkage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel ifort</td>
<td>-fPIC</td>
<td>-shared -Wl,-soname lib&lt;name&gt;.so.X</td>
</tr>
<tr>
<td>Gfortran</td>
<td>-fPIC</td>
<td>-shared -Wl,-soname=lib&lt;name&gt;.so.X</td>
</tr>
<tr>
<td>PGI pgf90</td>
<td>-fPIC</td>
<td>-shared -Wl,-soname=lib&lt;name&gt;.so.X</td>
</tr>
<tr>
<td>NAG nagfor</td>
<td>-PIC</td>
<td>-shared -Wl,-soname=lib&lt;name&gt;.so.X</td>
</tr>
<tr>
<td>IBM xlf</td>
<td>-G -qpic=big</td>
<td>-qmkshrobj</td>
</tr>
<tr>
<td>Cray ftn</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Example (Intel compiler):

```plaintext
ifort -c -fPIC mylib.f90
ifort -o libmylib.so.1.0.0 -shared -Wl,-soname libmylib.so.1 mylib.o
```

Add symbolic links (Linux):

```plaintext
ln -s libmylib.so.1.0.0 libmylib.so.1
ln -s libmylib.so.1 libmylib.so
```
Linking against the library and running the executable

**Linux linkage:**
- specify **directory** where shared library resides
- specify **shorthand** for library name

**Execute binary**
- set **library path**
- **execute** as usual

```sh
ifort -o myprog.exe myprog.f90 -L../lib -lmylib
export LD_LIBRARY_PATH=\$HOME/lib:$LD_LIBRARY_PATH
./myprog.exe
```

- **note:** if both a static and a shared library are found, the shared library will be used by default
- there usually exist compiler switches which enforce static linking

- **note:** */etc/ld.so.conf* contains library paths which are always searched
- there usually exist possibilities to hard-code the library path into the executable

**libmylib.so** lives there

**don't need to set LD_LIBRARY_PATH in these two cases**
Special linkage options

- The `-Wl,` option can be used to pass options to the linker
  
- **Example 1:**
    - want to specify that a certain library `-lspecial` should be linked statically, others dynamically
    - this is not uniquely resolvable from the library specification if both static and dynamic versions exist!

```
ifort -o myprog.exe myprog.f90 -Wl,-static -L/special_path \ 
   -lspecial -Wl,-dy -L../lib -lmylib
```

- **Example 2:** hard-code path into binary

```
ifort -o myprog.exe myprog.f90 -Wl,-rpath -L../lib \ 
   -L../lib -lmylib
```

- avoids the need to set `LD_LIBRARY_PATH` before execution
Dynamic loading (1)

- **Supported by C library:**
  - open a shared library at run time
  - extract a symbol (function pointer)
  - execute function
  - close shared library

- **Small Fortran module `dlfcn`**
  - type definition `dlfcn_handle`
  - procedures `dlfcn_open()`, `dlfcn_symbol()`, `dlfcn_close()`

  **Note:** the result of `dlfcn_symbol()` is of type `c_funptr` to enable conversion to an explicit interface procedure procedure pointer

- **From Fortran**
  - usable via C interoperability and pointers to procedures
  - implement plug-ins

---

© 2009-19 LRZ
Advanced Fortran Topics - LRZ section 271
**Dynamic loading (2): An example program**

**Shared library `libset1.so`:**
- **BIND(C) procedure**

```fortran
use dlfcn
implicit none
abstract interface
  subroutine set(i) bind(c)
    integer, intent(inout) :: i
  end subroutine
end interface
integer :: i, istat
type(dlfcn_handle) :: h
type(c_funptr) :: cp
procedure(set), pointer :: fp
h = dlfcn_open('./libset1.so', &
               RTLD_NOW)
cp = dlfcn_symbol(h, 'set_val')
call c_f_procpointer(cp, fp)
i = 1
call fp(i)
ismat = dlfcn_close(h)
```

**Module procedure:**
- **explicit name mangling needed**

```fortran
h = dlfcn_open('./libset2.so', &
               RTLD_NOW)
! at most one line valid
cp = dlfcn_symbol(h, &
                  '__s_MOD_set_val')
cp = dlfcn_symbol(h, &
                  's_mp_set_val')
cp = dlfcn_symbol(h, &
                  '__s_NMOD_set_val')
cp = dlfcn_symbol(h, &
                  's_MP_set_val')
call c_f_procpointer(cp, fp)
i = 1
call fp(i)
ismat = dlfcn_close(h)
```

**nm `libset2.so` | grep -i set_val**

- gfortran
- ifort
- xlf
- NAG, g95
- OK with TS 29113
Wrapping up

This concludes the LRZ part of this course

Following now: Exercise session 12