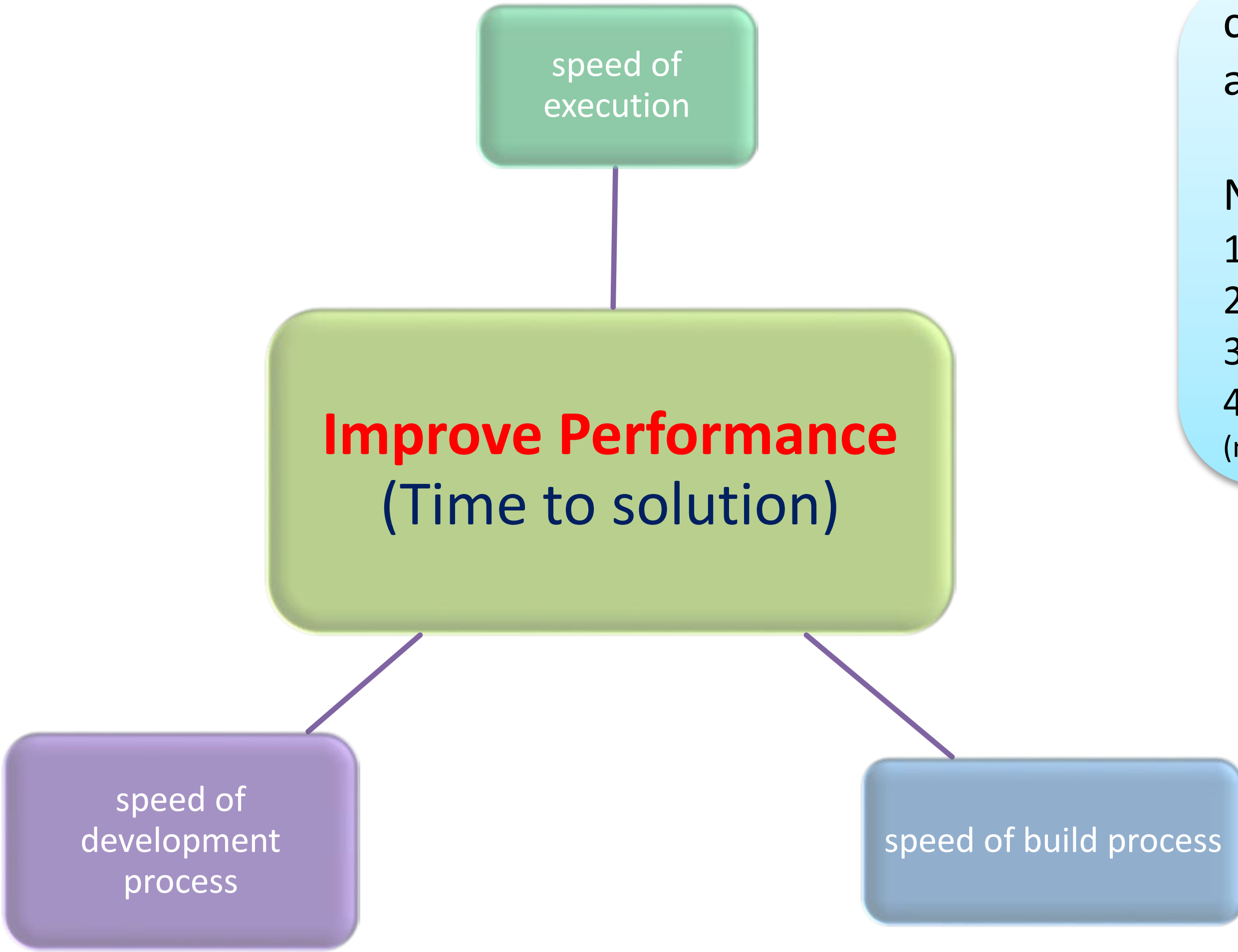
Fortran code modernization

Dr. Reinhold Bader
Leibniz Supercomputing Centre

Workshop's aims



correlations and anticorrelations exist

Need:

1. experience
2. compromise
3. intelligence
4. diligence

(not necessarily in that order)

How can the aims be achieved?

language features
- replace obsolescent / unsuitable features by modern ones
- follow best practices in using advanced features

correctness of code contributes to development speed

tools
- edit, document
- build, debug, profile, tune

data handling
- I/O processing and its design
- visualization

parallelism
- scalability in multiple facets
- proper choice of programming model

algorithms
- reduce problem complexity order while maintaining efficiency of execution

expect tradeoff

Assumptions on Audience

- **Good working knowledge of Fortran 77 semantics**
- **Knowledge about the most relevant Fortran 90/95 concepts**
  - modules, array processing, dynamic memory
- **Basic experience with C programming**
- **Basic experience with parallel programming**
  - using OpenMP, MPI or both
- **Useful:**
  - some conceptual knowledge about object-oriented programming (single inheritance, virtual methods, interface classes)

Assumptions on pre-existing code

- **Language features are used that**
  - date from Fortran 77 or earlier
  - were never standardized, but are supported in many compilers

- **How you proceed depends on the specifics of code reuse:**
  - run without (or at most minor isolated) modifications as a standalone program → no refactoring required

    „never change a running system“ + Fortran (mostly) backward compatible

  - use as library facility → no full refactoring may be needed, but it is likely desirable to create explicit interfaces
  - further maintenance (bug fixes with possibly non-obvious effects) or even further development is needed → refactoring is advisable

History of Fortran

Fortran – the oldest portable programming language

- first compiler developed by John Backus at IBM (1957-59)
- design target: generate code with speed comparable to assembly programming, i.e. for **efficiency** of compiled executables
- targeted at **scientific / engineering** (high performance) computing

Fortran standardization

- ISO/IEC standard 1539-1
- repeatedly updated

Generations of standards

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

TS → Technical Specifications

- „mini-standards“ targeted for future inclusion (modulo bug-fixes)

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

Legacy code


Recommend replacement by a more modern feature

OBS obsolescent feature

DEL deleted feature

Implementation dependencies

Processor dependent behaviour (may be unportable)

Performance


language feature for / against performance

Why Fortran?

SW engineering aspects

- good ratio of learning effort to productivity
- good optimizability
- compiler correctness checks

  (constraints and restrictions)

Ecosystem

- many existing legacy libraries
- existing scientific code bases → may determine what language to use
- using tools for diagnosis of correctness problems is sometimes advisable

Key language features

- dynamic (heap) memory management since F95, much more powerful in F03
- encapsulation and code reuse via modules since F95
- object based F95 and object-oriented F03 features
- array processing F95
- versatile I/O processing
- abstraction features: overloaded and user-defined operators F95
- interoperability with C F03 F18
- FP exception handling F03
- parallelism F08 F18

When not to use Fortran

- **When programming an embedded system**
  - these sometimes do not support FP arithmetic
  - implementation of the language may not be available
- **When working in a group/project that uses C++, Java, Eiffel, Haskell, … as their implementation language**
  - synergy in group: based on some – usually technically justified – agreement
  - minor exception: library code for which a Fortran interface is desirable – use C interoperability features to generate a wrapper

Some references

- **Modern Fortran explained** (8th edition incorporates F18)
  - Michael Metcalf, John Reid, Malcolm Cohen, OUP, 2018
- **The Fortran 2003 Handbook**
  - J. Adams, W. Brainerd, R. Hendrickson, R. Maine, J. Martin, B. Smith. Springer, 2008
- **Guide to Fortran 2008 programming** (introductory text)
  - W. Brainerd. Springer, 2015
- **Modern Fortran – Style and Usage** (best practices guide)
  - N. Clerman, W. Spector. Cambridge University Press, 2012
- **Scientific Software Design – The Object-Oriented Way**
  - Damian Rouson, Jim Xia, Xiaofeng Xu, Cambridge, 2011

References cont'd

- **Design Patterns – Elements of Reusable Object-oriented Software**
  - E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison-Wesley, 1994
- **Modern Fortran in Practice**
  - Arjen Markus, Cambridge University Press, 2012
- **Introduction to High Performance Computing for Scientists and Engineers**
  - G. Hager and G. Wellein

Dealing with legacy language features

Legacy code: Fixed source form

- **Source code stored in files with extension**

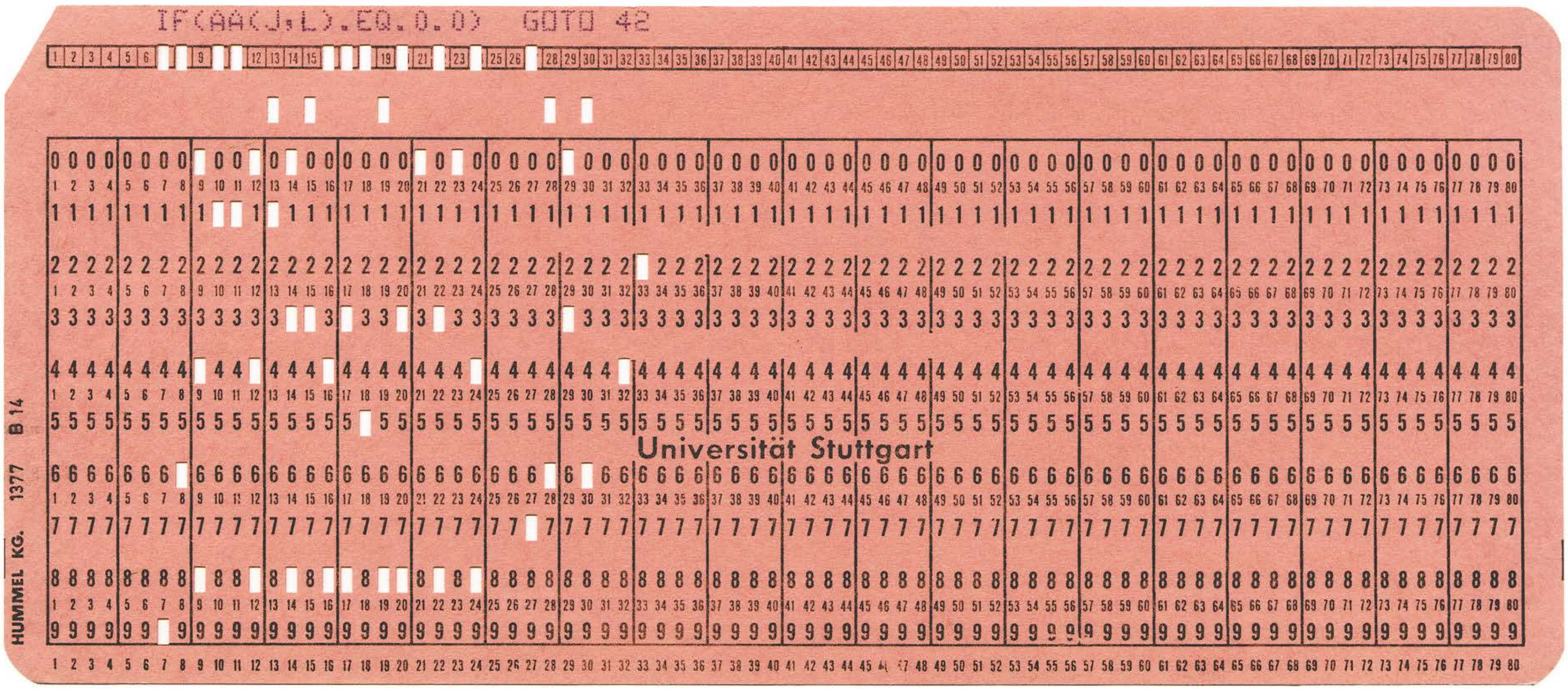
.f .for .ftn .F

for use with C-style preprocessing

- **Layout of code looks something like this**

```
C        1         2         3         4         5         6         7         8
*2345678901234567890123456789012345678901234567890123456789012345678901234567890
      PROGRAM M
        Y = 1.0
        X = 1.5
        X = X + 2.0                                                     +Y
        IF (X < 4.0) GOTO 20
        WRITE(*,*) 'statement with continuation',
     1             'line', X
C    comment line
   20   CONTINUE
      END PROGRAM
```

Technical reason for fixed source form …



Legacy code: Fixed source form

- **Fortran 77** **(and earlier)** **language rules for layout:**
  - statements must start at column 7
  - must end at column 72
  - continuation line: single non-blank / non-zero character in column 6
  - limit of 19 continuation lines
  - comment must have the characters C or * in column 1
  - labels must be in columns 1-5

processor extensions to 132 columns exist. **Beware** unnoticed errors if compiler option is not toggled

```
C         1         2         3         4         5         6         7         8
*2345678901234567890123456789012345678901234567890123456789012345678901234567890
      PROGRAM M
        Y = 1.0
        X = 1.5
        X = X + 2.0                                                    +Y
        IF (X < 4.0) GOTO 20
        WRITE(*,*) 'statement with continuation',
     1             'line', X
C    comment line
   20   CONTINUE
      END PROGRAM
```

might get ignored if line length extension is not toggled

Further pitfall: Insignificance of embedded blanks

```
      S = 0.0
      DO 10 I=1.5
        S = S+I
   10 CONTINUE
      WRITE(*,*) 'S=',S
      END
```

```
      C = 0.0
      AA = 2.0
      BB = 2.0
      IF (AA .EQ. BB) THEN C = AA + BB
      WRITE(*,*) 'C=',C
      END
```

Both codes are conforming, but deliver results that might surprise you ...

Quiz: Which language feature conspires with the embedded blanks to produce this surprise?

The new way: Rules for free source form

Program line

- upper limit of **132** characters
- arbitrary indentation allowed

Continuation line

- indicated by ampersand:

```
WRITE(*,fmt=*) &
  'Hello'
```

- variant for split tokens:

```
WRITE(*,fmt=*) 'Hel&
   &lo'
```

- upper limit: **255**

Multiple statements

- semicolon used as separator

```
a = 0.0; b = 0.0; c = 0.0
```

Comments:

- after statement on same line:

```
WRITE(*,*) 'Hello' ! produce output
```

- separate comment line:

```
WRITE(*,*) 'Hello'
! produce output
```

The art of commenting code:

- concise
- informative
- non-redundant
- consistent

  (maintenance issue)

File extension

.f90 .F90

- unrelated to language level

Tooling options

- **Open-source software**
  - `convert` tool by Michael Metcalf
  - `to_f90` tool by Alan Miller
  - your mileage may vary
  - further similar tools exist

- **NAG compiler**
  - supports `=polish` as an option for converting between fixed and free format
  - additional suboptions are available

Implicit and explicit typing of variables (1)

- **If no type declaration statement appears:**

  without an `IMPLICIT` statement, typing of entities is performed implicitly, based on first letter of the variable's name:

  - `a,...,h` and `o,...,z` become default real entities
  - `i,...,n` become default integer entities

- **Example program:**

```
PROGRAM declarations
  REAL :: ip
  xt = 5    ! xt is real
  i = 2.5   ! i is integer
  ip = 2.5 ! ip is real
  WRITE(*,*) x, i, ip
END PROGRAM
```

- **Note:**
  - newer (scripting) languages perform auto-typing by **context**
  - this is not possible in Fortran

Implicit and explicit typing of variables (2)

Modify implicit typing scheme

- IMPLICIT statement:

```
IMPLICIT DOUBLE PRECISION (a-h,o-z)
```

  changes implicitly acquired type for variables starting with letters `a-h,o-z` and leaves default rules intact for all other starting letters

- quite commonly used for implicit precision advancement

```
PROGRAM AUTO_DOUBLE
  IMPLICIT DOUBLE PRECISION (a-h,o-z)

  xt = 3.5 ! xt has extended precision
  :
END PROGRAM
```

The RHS constant is still single precision
→ loss of digits is possible

Recommendation:

- enforce strong typing with `IMPLICIT NONE`
- programmer is obliged to explicitly declare all variable‘s types

Legacy notations for intrinsic types

- **The following never was supported in any standard**
  - but is supplied as an extension by many implementations

```
INTEGER*4 JJ
INTEGER*8 JJEXT
REAL*4    X
REAL*8    XEXT
```

  - the parametrization refers to the number of bytes of storage needed by a scalar entity of the type
- **Compiler options for default type promotion** (e.g., `-i8, -r8`)
  - can have unforeseen side effects → avoid use of these
  - note that the standard requires default integers and reals to use the same number of numeric storage units
- **Recommendation**
  - replace declarations with appropriate KIND parameters for the type in question

Intrinsic numeric types – KIND parameterization

Declarations that should always work

- by virtue of standard's prescriptions:

```
INTEGER, PARAMETER :: ik = KIND(0), &
                      lk = SELECTED_INT_KIND(18), &
                      rk = KIND(1.0), &
                      dk = SELECTED_REAL_KIND(10,37)

INTEGER(ik)       :: jdefault  ! default integer, can represent 10^5
INTEGER(KIND=lk) :: jlarge    ! can represent 10^18

REAL(rk)          :: xdefault  ! default real,
                               ! likely at least 6 digits
REAL(dk)          :: xdouble   ! likely double precision,
                               ! at least 10 digits
```

compile time constant → unmodifiable

digits

decimal exponent

2-colon separator improves readability

- FP numbers as declared above will **usually** use IEEE-754 conforming representations (no guarantee, but in the following this will be assumed)
- the KIND values themselves are **not** portable

Numeric models for integer and real data

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

$$x = b^e \times s \times \underbrace{\sum_{k=1}^{p} f_k \times b^{-k}}_{\text{fractional part}} \quad \text{or } \mathbf{x = 0}$$

- **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 $w_k \in \{0, ..., r\text{-}1\}$

base 2 → „Bit Pattern"

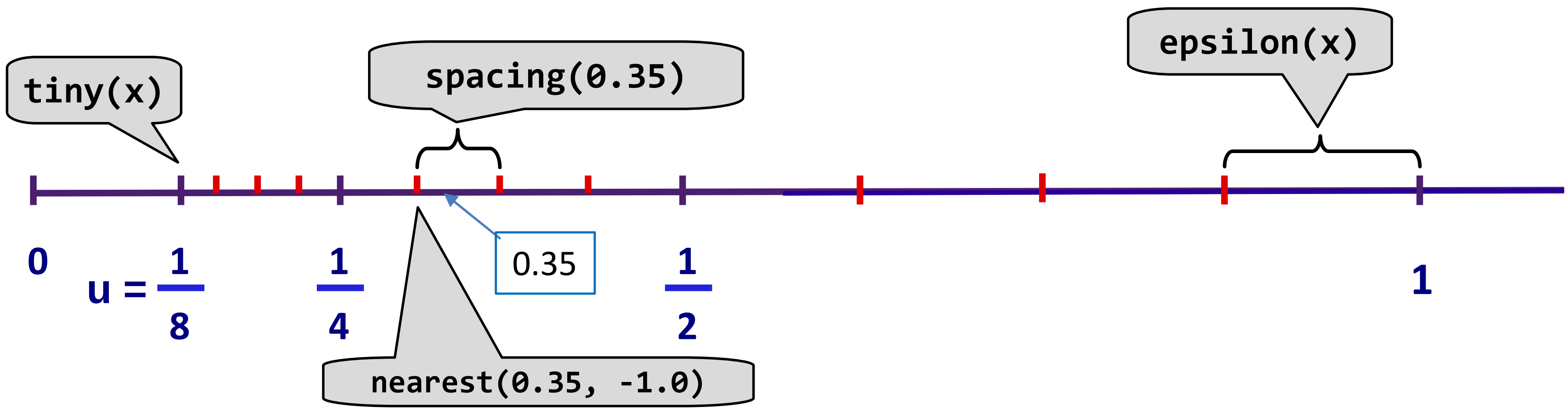
- **real kind is defined by**
  - positive integers p (digits), b > 1 (base, normally b = 2)
  - integers $e_{min} < e_{max}$
- **real value is defined by**
  - sign $s \in \{\pm 1\}$
  - integer exponent $e_{min} \leq e \leq e_{max}$
  - sequence of $f_k \in \{0, ..., b\text{-}1\}$, f_1 nonzero

Inquiry intrinsics for model parameters

`digits(x)`	for real oder integer x, returns **the number of digits** (p, q respectively) as a default integer value.	`minexponent(x)`, `maxexponent(x)`	for real x, returns the default integer $\mathbf{e_{min}}$, $\mathbf{e_{max}}$ respectively
`precision(x)`	for real or complex x, returns the default integer indicating the **decimal precision** (=decimal digits) for numbers with the kind of x.	`radix(x)`	for real or integer x, returns the default integer that is the **base** (b, r respectively) for the model x belongs to.
`range(x)`	for integer, real or complex x, returns the default integer indicating the **decimal exponent range** of the model x belongs to.		

Inquiry intrinsics for model numbers

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



`rrspacing(x) = abs(x) / spacing(x)`

- largest representable number: $^7/_2$ (beyond that: **overflow**) — `huge(x)`

Mapping fl: $\mathbb{R} \ni x \to fl(x)$

- to nearest model number
- maximum relative error

$$fl(x) = x \cdot (1 + d), | d | < u$$

F03

IEEE facilities

- **Special intrinsic modules exist**
  - enable use of IEEE-conforming representations
  - enforce use of IEEE-conforming floating point operations
  - deal with special values (subnormals, infinities, NaNs)
  - deal with rounding by proper use of rounding modes
  - many module procedures
- **Exception handling**
  - five floating point exceptions (underflow, overflow, division by zero, invalid, inexact)
  - run-time dispatch vs. termination (halting)
  - save and restore floating point state
- **Tiresome to use ...**
  - only at (few) critical locations in application
  - if a (slow) fallback is needed in case a fast algorithm fails

Inquiry intrinsics for real and integer types

(courtesy Geert Jan Bex, using Intel Fortran)

default real — double precision

	REAL32	REAL64	REAL128
HUGE	3.40282347E+38	1.7976931E+308	1.1897315E+4932
TINY	1.17549435E-38	2.2250739E-308	3.3621031E-4932
EPSILON	1.19209290E-07	2.2204460E-016	1.9259299E-0034
RANGE	37	307	4931
PRECISION	6	15	33

default integer

	INT8	INT16	INT32	INT64
HUGE	127	32767	2147483647	9223372036854775807
RANGE	2	4	9	18

Notes

- `REAL32, ..., INT8, ...` are KIND numbers defined in the `ISO_FORTRAN_ENV` intrinsic module
- numbers refer to storage size in bits
- if two KINDs using 32 bits exist, `REAL32` might be different from default real

Notation for operators

- **Modern Fortran is more readable**

F77	F95	Meaning
.LT.	<	less than
.LE.	<=	less than or equal
.EQ.	==	equal
.NE.	/=	not equal
.GT.	>	greater than
.GE.	>=	greater than or equal

Non-numeric intrinsic types

Character type

legacy code often has `CHARACTER*11`

```
CHARACTER :: ch                          ! a single default character
CHARACTER(LEN=11) :: str                 ! length-parametrization
                                         ! supplies fixed-length strings
ch  = 'p'
str = 'Programming'
str = str(5:7) // ch                     ! result is 'ramp       '
```

not very flexible, but we'll learn a better way soon

- principle of least surprise (blank padding, truncation)
- UNICODE support is possible via (non-default) KIND

Logical type

```
LOGICAL :: switch                        ! default logical flag

switch = .TRUE.                          ! or .FALSE.
switch = (i > 5) .neqv. switch           ! logical expressions
                                         ! and operators
```

F03

C-interoperable intrinsic types
(relative to companion C processor)

A subset of KIND parameter values

- defined in the `ISO_C_BINDING` intrinsic module
- unsigned types are not supported

C type	Fortran declaration	C type	Fortran declaration
int	INTEGER(c_int)	char	CHARACTER(**LEN=1**,KIND=c_char)
long int	INTEGER(c_long)		
size_t	INTEGER(c_size_t)		
[un]signed char	INTEGER(c_signed_char)	_Bool	LOGICAL(c_bool)
float	REAL(c_float)		
double	REAL(c_double)		

likely the same as `kind('a')`

may be same as `c_int`

On x86 architecture: the same as default real/double prec. type. But this is not guaranteed.

- 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

Complex types

- **Not as heavily used as floating point numbers, but still ...**

```
USE, INTRINSIC :: iso_c_binding
IMPLICIT NONE
INTEGER, PARAMETER :: rk = KIND(1.0), &
                      dk = SELECTED_REAL_KIND(10,37)

COMPLEX(rk)       :: cdefault        ! default COMPLEX,
COMPLEX(dk)       :: cdouble         ! double precision COMPLEX
COMPLEX(c_float_complex) :: cc       ! can interoperate with
                                     ! C99 float _Complex

cdouble = (1.0_dk, 2.5_dk)           ! 1 + 2.5 i
```

- two numeric storage units per variable for default complex
- four numeric storage units for double complex

Legacy control flow: Branching via the GO TO statement

Transfer of control

```
    GO TO 100
 80 x = x + 3.0
    :
100 x = x * 2.0
    IF (…) GO TO 80
```

unconditional execution (GO TO 100)

conditional branch backward (IF (…) GO TO 80)

- argument is a label
- regular execution is resumed at correspondingly labeled statement (in same program unit)

Risks:

- dead code (often removed by compiler)
- subtle bugs in control flow that cause infinite looping or incorrect results
- code often hard to understand and maintain

Recommendation:

- Avoid using this statement if any other block construct can do the job
- Examples follow ...

Conditional execution of statements (1)

Arithmetic IF — DEL

```
  IF (expr) 2, 7, 8
2 … ! expr < 0
  GO TO 10
7 … ! expr == 0
  GO TO 10
8 … ! expr > 0
10 CONTINUE
```

do-nothing statement

- `expr` can be integer or real
- note that additional GOTOs are usually needed
- can also set up two-way branch (how?)

IF block construct — F95

```
IF (expr < 0) THEN
  … ! expr < 0
ELSE IF (expr == 0) THEN
  … ! expr == 0
ELSE
  … ! expr > 0
END IF
```

- might need special treatment for overlapping execution (fall-through)

Conditional execution of statements (2)

Computed GOTO

OBS

- evaluate integer expression

```
   GO TO (2, 7, 8) expr
   … ! expr < 1 or > 3
   GO TO 10
2  … ! executed if expr == 1
   GO TO 10
7  … ! executed if expr == 2
   GO TO 10
8  … ! executed if expr == 3
10 CONTINUE
```

- again: beware overlapping execution

SELECT CASE construct

F95

```
SELECT CASE (expr)
CASE (1)
  …
CASE (2)
  …
CASE (3)
  …
CASE default
  …
END SELECT
```

- easier to read and understand

Arrays

Declaration

```
INTEGER, PARAMETER :: &
   ndim = 2, mdim = 3, kdim = 9

REAL(rk)  :: c(ndim, mdim), &
             d(0:mdim-1, kdim), &
             a(ndim), b(mdim)
```

- here: rank 2 and rank 1 arrays (up to rank 15 is possible)
- default lower bound is 1

Purpose

- efficient large scale data processing

Dynamic sizing?

- supported F95 F03

Array constructor F95

```
c = RESHAPE( &
        [ (REAL(i,rk),i=1,6) ], &
          SHAPE(c) )
```

- constructor `[ ]` F03 or `(/ /)` generates rank 1 arrays only
- use intrinsic functions to query or change the shape
- use implicit do loops to generate large arrays

Sectioning

```
d(0::2,1:kdim:3) = c
```

- array subobject created by subscript triplet specification
- array syntax for assignment

Attributes

- **General concept:**
  - declare an additional property of an object
- **Example:**
  - DIMENSION attribute: declares an object to be an array
  - as an implicit attribute

```
REAL(rk) :: a(ndim)
```

(this is particular to DIMENSION, though)

- **Syntax for attributes:**
  - may appear in attribute form or in statement form
  - attribute form

```
REAL(rk), DIMENSION(ndim) :: a
```

  - statement form

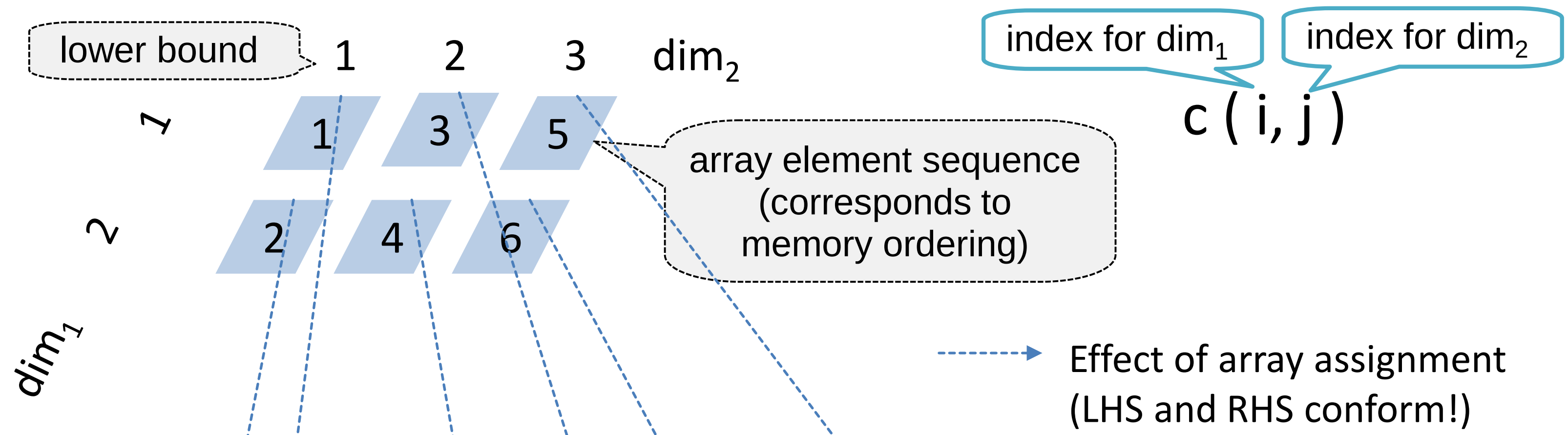
```
REAL(rk) :: a
:
DIMENSION(ndim) :: a
```

(not recommended, because non-local declarations are more difficult to read)

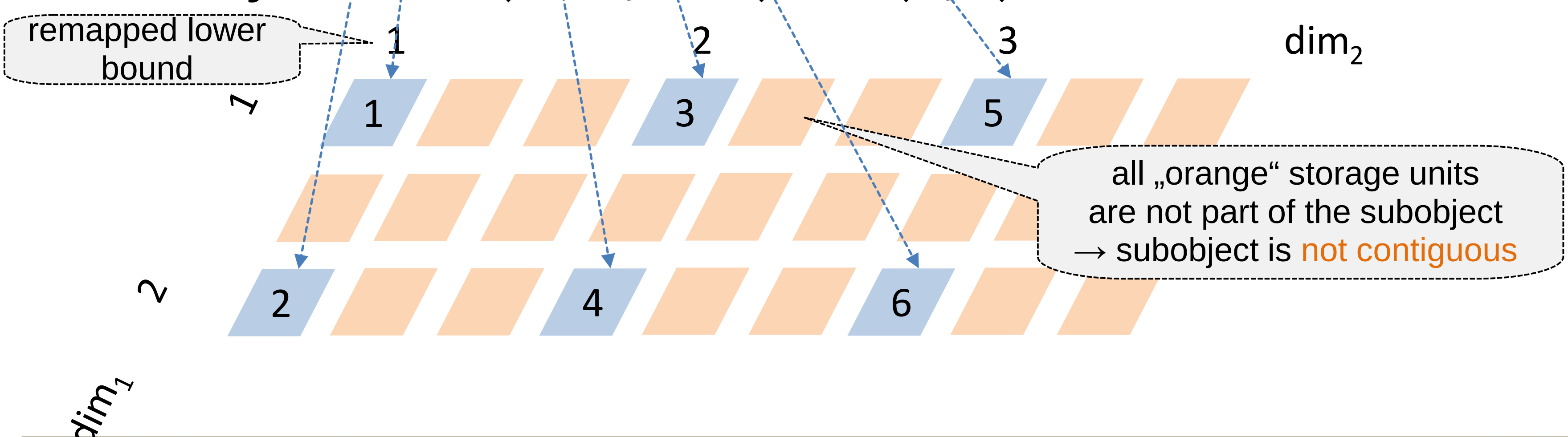
The three declarations of entity „a“ on this slide are semantically equivalent

Array storage layout

- **Element ordering: column major**



- **Array section** `d(0::2,1::3)` **of** `d(:,:)`



Legacy versions of looping (1)

OBS Non-block DO loop

- use a statement label to identify end of construct

```
   DO 20 k = 1, mdim
     IF (…) GO TO 20
     DO 10, j = 1, ndim
       c(j, k) = a(j) * b(k)
10   CONTINUE
20 CONTINUE
```

- nested loops require separate labeled statements
- use is not recommended

DEL Shared loop termination statement

```
DO 10 k=1, mdim
  : ! (X)
  DO 10 j=1, ndim
10  c(j, k) = a(j) * b(k)
```

- 200 loop iterations including execution of labeled statement
- notation is confusing
- statement (X) of form

```
IF (…) GO TO 10
```

is not permitted because label is considered to belong to inner loop

Modern DO Loop Construct
(with fine-grain execution control)

Block construct

- for finite looping

```
outer : DO k = 1, mdim
  IF (…) CYCLE outer
  inner : DO j = 1, ndim
    c(j, k) = a(j) * b(k)
  END DO inner
END DO outer
```

- Optional naming of construct
- CYCLE skips an iteration of the specified loop
  (default: innermost loop)
- strided loops also allowed

Unknown iteration count

```
iter : DO
  :
  IF (diff < eps) EXIT iter
  :
END DO iter
```

- EXIT terminates specified block construct (this also works for non-loop constructs)
- Alternative:

```
DO WHILE (diff >= eps)
  :
END DO
```

Legacy versions of looping (2)

Non-integer loop variable

```
REAL :: r, s, stride
s = 0.0
stride = 1.0000001
DO r = 1.2,10.2,stride
  s = s + r
END DO
```

- borderline cases where number of iterations may depend on implementation, rounding etc.

Replace by integer loop variable

```
REAL :: r, s, stride
INTEGER :: ir
s = 0.0
stride = 1.0000001
r = 1.2
DO ir = 1,9
  r = r + stride
  s = s + r
END DO
```

- numerics may still be questionable …

Concept of derived type

Overcome insufficiency

- of intrinsic types for description of abstract concepts

```
MODULE mod_body
 :
 TYPE :: body
   CHARACTER(LEN=4) :: units
   REAL :: mass
   REAL :: pos(3), vel(3)
 END TYPE body
CONTAINS
  …
END MODULE
```

declarations of **type components**

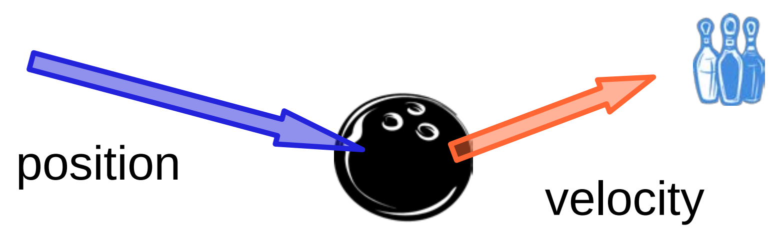
Formal type definition

Recommendation:

- a derived type definition should be placed in the specification section of a **module**.

a program unit introduced by Fortran 90

Reason: it is otherwise not reusable (simply copying the type definition creates a second, distinct type)



Type components:

- can be of intrinsic or derived type, scalar or array
- further options discussed later

layered creation of more complex types from simple ones

Structures

- **Objects of derived type**
- **Examples:**

access type from outside `mod_body`

```
USE mod_body
TYPE(body) :: ball, copy
TYPE(body) :: asteroids(ndim)
```

- creates two scalars and an array with ndim elements of `type(body)`
- sufficient memory is supplied for all component subobjects
- access to type definition here is by use association

- **Structure constructor**
  - permits to give a value to an object of derived type (complete definition)

```
ball = body( 'MKSA', mass=1.8, pos=[ 0.0, 0.0, 0.5 ], &
                     vel=[ 0.01, 4.0, 0.0 ] )
```

  - It has the same name as the type,
  - and keyword specification inside the constructor is optional.
    (you must get the component order right if you omit keywords!)

- **Default assignment** `copy = ball`
  - copies over each type component individually

Structures as dummy arguments

Implementation of „methods“

```
MODULE mod_body
  :
  TYPE :: body
  …
CONTAINS
   SUBROUTINE kick(this, …)
     TYPE(body), intent(inout) :: this
     …
   END SUBROUTINE
END MODULE
```

type definition shown earlier

- declares scalar dummy argument of `type(body)`
- access to type definition here is by host association

```
USE mod_body
TYPE(body) :: ball
TYPE(body) :: asteroids(ndim)
… ! define objects
CALL kick(ball, …)
CALL kick(asteroids(j), …)
```

- invocation requires an actual argument of exactly that type
  (→ explicit interface required)

Accessing type components

- **Via selector** `%`

```
SUBROUTINE kick(this, dp)
  TYPE(body), INTENT(inout) :: this
  REAL, INTENT(in) :: dp(3)
  INTEGER :: i

  DO i = 1, 3
    this % vel(i) = this % vel(i) + dp(i) / this % mass
  END DO
END SUBROUTINE
```

- `this % vel` is an array of type real with 3 elements
- `this % vel(i)` and
  `this % mass` are real scalars

(spaces are optional)

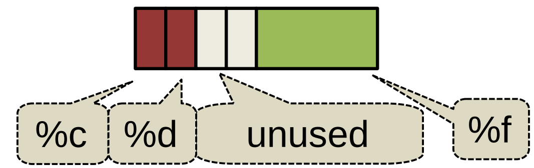
Remarks on storage layout

Single derived type object

- compiler might insert padding between type components

```
TYPE :: d_type
  CHARACTER :: c
  REAL :: f
  CHARACTER :: d
END TYPE
```

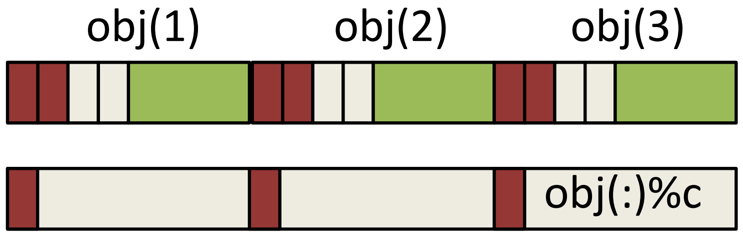
storage layout of a `TYPE(d_type)` scalar object could look like



Array element sequence

- as for arrays of intrinsic type

```
TYPE(d_type) :: obj(3)
```



Legacy features and extensions

Sequence types

- **enforce** storage layout in specified order

```
TYPE :: s_type
  SEQUENCE
  REAL :: f
  INTEGER :: il(2)
END TYPE
```

Note:

- usability of sequence types is restricted
- no type parameters, non-extensible

- multiple type declarations with same type name and component names are permitted

Structures

- non-standard syntax for derived types, pre- F95 . Semantics are the same.

```
STRUCTURE /body/
  REAL mass
  REAL pos(3)
  REAL vel(3)
END STRUCTURE
```

```
! object
RECORD /body/ ball
```

C-interoperable derived types

BIND(C) types

- **enforce** C struct storage layout:

```
USE, INTRINSIC :: iso_c_binding

TYPE, BIND(C) :: c_type
  REAL(c_float) :: f
  INTEGER(c_int) :: il(2)
END TYPE
```

is interoperable with

```
typedef struct {
  float s;
  int i[2];
} Ctype;
```

Note:

- usability of BIND(C) types is restricted
- no type parameters, non-extensible

Procedures and their interfaces

Subprogram invocation: Fortran 77 style implicit interface

- **Simple example:** solve $ax^2 + bx + c = 0$

```
      SUBROUTINE solve_quadratic (a, b, c, n, x1, x2)

      IMPLICIT NONE
      REAL a, b, c, x1, x2
      INTEGER n
C declare local variables
      :
C calculate solutions
      :
      END SUBROUTINE
```

(a, b, c, n, x1, x2): **dummy argument** list

- usually stored in a separate file → „**external procedure**“
- commonly together with other procedures (solve_linear, solve_cubic, ...)

Invoking a procedure with implicit interface

- **Unsafe legacy usage**

```
      PROGRAM q_implicit
      IMPLICIT NONE
      REAL a1, a2, a3, x, y
      INTEGER nsol
      EXTERNAL solve_quadratic
C initialize a1, a2, a3
      :
      CALL solve_quadratic( a1, a2, a3, nsol, x, y )
      WRITE(*, *) nsol, x, y
      END PROGRAM
```

Often forgotten. Most relevant in case of name collision with an intrinsic.

actual argument list

- **Disadvantages:**
  - compiler cannot check on correct use of number, type, kind and rank of arguments (**signature** or **characteristics** of interface)
  - many features of modern Fortran cannot be used at all (for example, derived type arguments, or assumed-shape dummy arrays, etc.)

Advice: Avoid implicit interfaces …

- **… by using one of the following cures:**

1. **Code targeted for future development:**
   Convert all procedures to module procedures
2. **Legacy library code that should not be modified:**
   Manual or semi-automatic creation of explicit interfaces for external procedures
   a. create include files that contain these interfaces, or
   b. create an auxiliary module that contains these interfaces

We'll look at each of these in turn on the following slides

1. Best method: Create module procedures

Implies an automatically created explicit interface

```
MODULE mod_solvers
  IMPLICIT NONE
CONTAINS
  SUBROUTINE solve_quadratic( a, b, c, n, x1, x2 )
    REAL :: a, b, c
    REAL :: x1, x2
    INTEGER :: n
     : ! declare local variables
     : ! calculate solutions
  END SUBROUTINE
  :
END MODULE
```

further procedures (solve_linear, solve_cubic, ...)

Invoking the module procedure

- **Access created interface via USE statement**

```
PROGRAM q_module
  USE mod_solvers
  IMPLICIT NONE
  REAL :: a1, a2, a3, x, y
  INTEGER :: nsol

  a1 = 2.0; a2 = 7.4; a3 = 0.2
  CALL solve_quadratic( a1, a2, a3, nsol, x, y )
  WRITE(*, *) nsol, x, y
END PROGRAM
```

access PUBLIC entities of mod_solvers by USE association

actual argument list

- compile-time checking of invocation against accessible interface

Invoking procedures with explicit interfaces

- **Argument association**
  - each dummy argument becomes associated with its corresponding actual argument
- **Invocation variants:**
  1. **Positional correspondence**

     ```
     CALL solve_quadratic( a1, a2, a3, nsol, x, y  )
     ```

     for the above example: **a ↔ a1, b ↔ a2, x2 ↔ y** etc.
  2. **Keyword arguments** → caller may change argument ordering

     ```
     CALL solve_quadratic( a1, a2, a3, x1=x, x2=y, n=nsol )
     ```

the Fortran standard does not specify the means of establishing the association

however, efficiency considerations usually guide the implementation
(avoid data copying wherever possible)

Remember the dependencies ...

- **Separate compilation**
  - different program units are usually stored in **separate** source files
- **Previous example** (assuming an intuitive naming convention)

```
gfortran -c -o mod_solvers.o mod_solvers.f90
```

also creates module information file `mod_solvers.mod`
→ must compile `q_module` **after** `mod_solvers`

```
gfortran -c -o q_module.o q_module.f90

gfortran -o main.exe q_module.o mod_solvers.o
```

compile sources to objects
(binary code, but not executable)

link objects into executable

- **Remember:**
  - module dependencies form a directed acyclical graph
  - changes in modules force recompilation of dependent program units
  - module information file: a precompiled header

2. Manual declaration of an interface block

(note that this is neither needed nor permitted for module procedures!)

```
INTERFACE
  SUBROUTINE solve_quadratic( a, b, c, n, x1, x2 )
    REAL :: a, b, c, x1, x2
    INTEGER :: n
  END SUBROUTINE
END INTERFACE
```

no executable statements

interface body

- specification syntax that describes the characteristics („signature") of the procedure. Provides an explicit interface for an external procedure
- some compilers/tools can generate interface blocks from source of external procedures via a switch (may be more reliable!)
- allows to avoid disadvantages of implicit interfaces if the interface block is accessible in the program unit that invokes the procedure

Handling interface blocks (2a.)

Variant a.

- place interface block in an include file, say `solvers.inc`
- the file might contain lots of interface blocks, or an interface block with multiple interface specifications

Usage in calling program unit:

```
PROGRAM q_include
  IMPLICIT NONE
  REAL :: a1, a2, a3, x, y
  INTEGER :: nsol
  INCLUDE 'solvers.inc'

  a1 = 2.0; a2 = 7.4; a3 = 0.2
  CALL solve_quadratic( a1, a2, a3, nsol, x, y )
  WRITE(*, *) nsol, x, y
END PROGRAM
```

Statement performs textual insertion.
File can be reused from any program unit.

compilation performance issues can arise for large scale use

Handling interface blocks (2b.)

- **Variant b.**
  - Insert into specification part of a „helper“ module:

```
MODULE mod_interfaces
  IMPLICIT NONE
  INTERFACE
    SUBROUTINE solve_quadratic( a, b, c, n, x1, x2 )
      REAL :: a, b, c, x1, x2
      INTEGER :: n
    END SUBROUTINE
  END INTERFACE
END MODULE
```

Again, you can add further interfaces here

- **Access by USE association in the calling program unit**
  - analogous to `q_module`
  - formal difference is that an external object must be linked in

Declaring INTENT for dummy arguments

Inform processor about expected usage

```
SUBROUTINE solve_quadratic( a, b, c, n, x1, x2 )
  REAL, INTENT(in) :: a, b, c
  REAL, INTENT(inout) :: x1, x2
  INTEGER, INTENT(out) :: n
  :
END SUBROUTINE
```

specify additional attribute

Semantics

- effect on both implementation and invocation

implies the need for **consistent** intent specification (fulfilled for module procedures)

specified intent	property of dummy argument
`in`	procedure must not modify the argument (or any part of it)
`out`	actual argument must be a variable; it becomes undefined on entry to the procedure
`inout`	actual argument must be a variable; it retains its definition status on entry to the procedure

Examples for the effect of INTENT specifications

- **Compile-time rejection of invalid code**
  - subroutine implementation:

```
REAL, INTENT(in) :: a
:
a = … ! rejected by compiler
```

  - subroutine usage:

rejected by compiler

```
CALL solve_quadratic( a, t, s, n, 2.0, x )
```

- **Compiler diagnostic (warning) may be issued**
  - e.g. if `INTENT(out)` argument is not defined in the procedure

- **Unspecified intent**

violations → run-time error if you're lucky

⚠ actual argument determines which object accesses are conforming

Passing arguments by value

F03

- **Use the VALUE attribute**
  - for dummy argument
- **Example:**

```
SUBROUTINE foo(a, n)
  IMPLICIT NONE
  REAL, INTENT(inout) :: a(:)
  INTEGER, VALUE :: n
  :
  n = n - 3
  a(1:n) = …
END SUBROUTINE
```

  - a local copy of the actual argument is generated when the subprogram is invoked
  - often needed for C-interoperable calls

- **General behaviour / rules**
  - local modifications are only performed on local copy – they never propagate back to the caller
  - argument-specific side effects are therefore avoided
    → VALUE can be combined with PURE
  - argument may not be INTENT(out) or INTENT(inout)

    INTENT(in) is allowed but mostly not useful

Functions – a variant of procedure

Example:

$$wsqrt(x,p) = \sqrt{1 - \frac{x^2}{p^2}} \text{ if } |x| < |p|$$

```
MODULE mod_functions
  IMPLICIT NONE
CONTAINS
  REAL FUNCTION wsqrt(x, p)
    REAL, INTENT(in) :: x, p
    :
    wsqrt = …
  END FUNCTION wsqrt
END MODULE
```

function result declaration

calculate function value and then assign to result variable

To be used in expressions:

```
USE mod_functions
:
x1 = 3.2; x2 = 2.1; p = 4.7
y = wsqrt(x1,p) + wsqrt(x2,p)**2
IF (wsqrt(3.1,p) < 0.3) THEN
  …
END IF
```

Notes:

- function result is **not** a dummy variable
- no CALL statement is used for invocation

Using a RESULT clause

- **Alternative syntax for specifying a function result**
  - permits **separate** declaration of result and its attributes

```
FUNCTION wsqrt(x, p) RESULT( res )
  REAL, INTENT(in) :: x, p
  REAL :: res
  :
  res = …
END FUNCTION wsqrt
```

  - the invocation syntax of the function is not changed by this
- **In some circumstances, use of a RESULT clause is obligatory**
  - for example, directly RECURSIVE functions

Functions declared PURE

- **Example:**

```
PURE FUNCTION wsqrt(x, p) RESULT( res )
  REAL, INTENT(in) :: x, p
  REAL :: res
  :
END FUNCTION wsqrt
```

certain things not allowed here …

- **Compiler ensures freedom from side effects, in particular**
  - all dummy arguments have INTENT(IN)
  - neither global variables nor host associated variables are defined
  - no I/O operations on external files occur
  - no STOP statement occurs
  - …

  → compile-time **rejection** of procedures that violate the rules

troublesome for debugging
→ temporarily remove the attribute

- **Notes:**
  - in contexts where PURE is not needed, an interface not declaring the function as PURE might be used
  - in the implementation, obeying the rules becomes programmer's responsibility if PURE is not specified

Subroutines declared PURE, etc.

- **For subroutines declared PURE, the only difference from functions is:**
  - all dummy arguments must have declared INTENT

- **Notes on PURE procedures in general:**
  - Use of the PURE property (in contexts where it is required) in an invocation needs an explicit interface
  - PURE is needed for invocations in some block constructs, or invocations from (other) PURE procedures
  - another motivation for the PURE attribute is the capability to execute multiple instances of the procedure in parallel without incurring race conditions. However, it **remains** the **programmer‘s responsibility** to exclude race conditions for the assignment of function values, and for actual arguments that are updated by PURE subroutines.

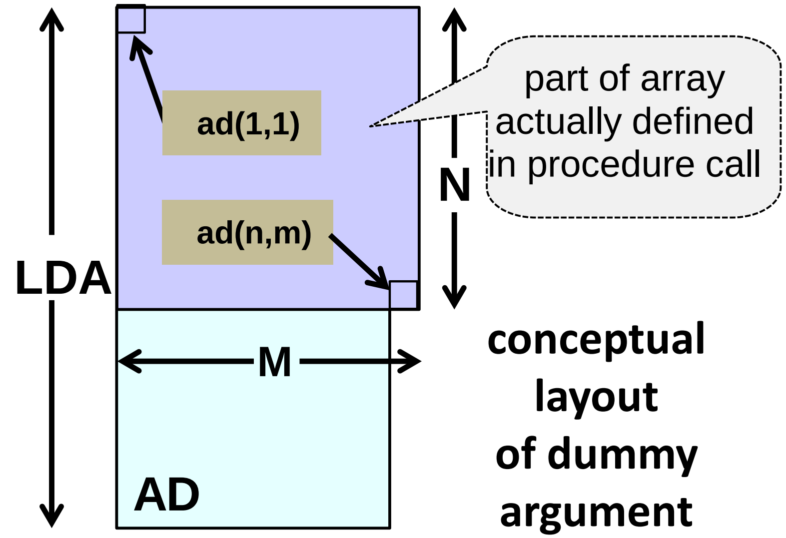
Assumed-size arrays: Typical interface design

(for use of legacy or C libraries)

```
SUBROUTINE slvr(ad, lda, n, m)
  INTEGER :: lda, n, m
  REAL :: ad( lda, * )
  …
  DO j=1, m
    DO i=1, n
      ad(i,j) = …
      …
    END DO
  END DO
  …
```

contiguous sequence of array elements

size is assumed from actual argument



- **Notes:**
  - leading dimension(s) of array as well as problem dimensions are explicitly passed
  - dummy argument does not have a shape and therefore cannot be defined or referenced as a whole array (sectioning is possible if a last upper bound is specified)
  - minimum memory requirement is implied by addressing: `LDA*(M-1) + N` array elements, where `N ≤ LDA`
  - Example: Level 2 and 3 BLAS interfaces (e.g., DGEMV)

Invocation variants

```
INTEGER, PARAMETER :: lda = …
REAL :: aa(lda, lda), ba(lda*lda)
: ! define m, n, …
```

```
CALL slvr( aa, lda, n, m )
```

```
CALL slvr( ba, lda, n, m )
```

```
CALL slvr( aa(i, j), lda, n, m )
```

```
CALL slvr(aa(1:2*n:2,:), n, n, m )
```

- **Permissible calls:**
  actual argument is a ...
  - complete or assumed-size array
    (indexing matches if done correctly)
  - array of differing rank
    (need to set up index mapping)
  - array element
    (work on a subblock, ad(1,1) ↔ aa(i,j))
  - non-contiguous array section
    (copy-in/out to an array temporary must be done by compiler)

- **Pitfalls:**
  - actual argument does not supply sufficient storage area
  - inconsistency of leading dimension specification
    e.g. „off-by-one“ → „staircase effect“

Explicit-shape dummy argument

Array bounds

- declared via non-constant specification expressions

```
SUBROUTINE slvr_explicit( &
                ad, lda, n, m)
  INTEGER :: lda, n, m
  REAL :: ad( lda, n )
  …
```

- also sometimes used in legacy interfaces ("adjustable-size array")
- in Fortran 77, a value of zero for n was not permitted

Argument passing

- works in the same way as for an assumed size object
- except that the dummy argument has a shape

(therefore the actual argument must have at least as many array elements as the dummy if the whole dummy array is referenced or defined)

Manually created interface for C library calls

- **Example: C function with prototype**

```
float lgammaf_r(float x, int *signp);
```

- **Fortran interface: the `BIND(C)` attribute**

Note: BIND(C) module procedures are also permissible

```
MODULE libm_interfaces
 IMPLICIT NONE
 INTERFACE
   REAL(c_float) FUNCTION lgammaf_r(x, is) BIND(C)
     USE, INTRINSIC :: iso_c_binding

     REAL(c_float), VALUE :: x
     INTEGER(c_int) :: is
   END FUNCTION
 END INTERFACE
END MODULE
```

enforce C name mangling

provides kind numbers for interoperating types

C-style value argument

Mixed-case C functions

- **An additional label is needed**

```
// example C prototype
void Gsub(float x[], int n);
```

invocation from Fortran via Fortran name

```
INTERFACE
  SUBROUTINE ftn_gsub(x, n) BIND(C, name='Gsub')
    USE, INTRINSIC :: iso_c_binding
    REAL(c_float), dimension(*) :: x
    INTEGER(c_int), value :: n
  END FUNCTION
END INTERFACE
```

  - a string constant denoting the case-sensitive C name
- **C-style arrays**
  - glorified pointers of interoperable type
  - require assumed size declaration in matching Fortran interface
- **Implementation may be in C or Fortran**
  - in the latter case, a BIND(C) module procedure can be written

Assumed shape dummy argument

This is the recommended array argument style

```
MODULE mod_solver
  IMPLICIT NONE
CONTAINS
  SUBROUTINE process_array(ad)
    REAL, INTENT(inout) :: ad(:,:)
    INTEGER :: i, j
    :
    DO j=1, SIZE(ad,2)
      DO i=1, SIZE(ad,1)
        ad(i,j) = …
        …
      END DO
    END DO
    :
  END SUBROUTINE
END MODULE
```

assumed shape
rank 2 array

Notes

- shape/size are **implicitly** available
- lower bounds are 1 (by default), or are explicitly specified, like

```
REAL :: ad(0:,0:)
```

Usage of the procedure

Invocation is straightforward

```
PROGRAM use_solver
  USE mod_solver
  IMPLICIT NONE
  REAL :: aa(0:1, 3), ab(0:2, 9)

  : ! define aa, ab
  CALL process_array( aa )
  CALL process_array( ab(0::2,1::3) )
  :
END PROGRAM
```

access **explicit** interface for `process_array`

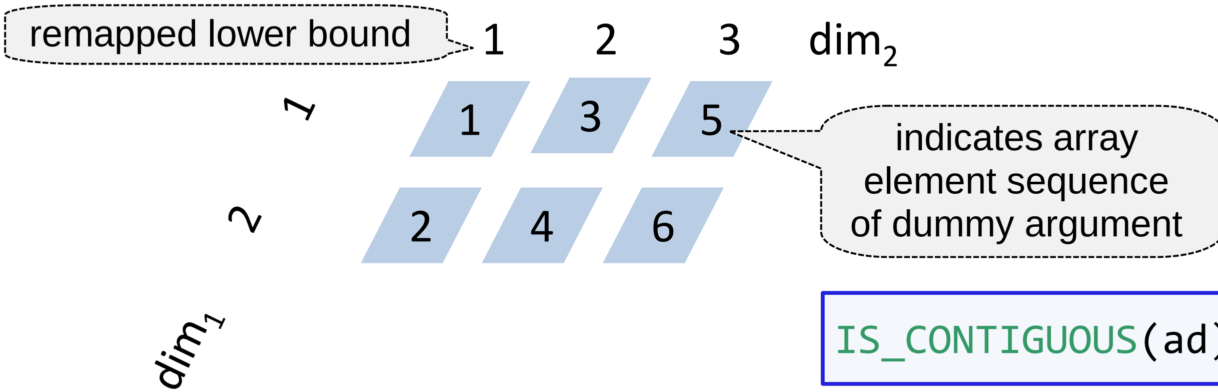
consistency of argument's **type, kind** and **rank** with interface specification is **required**

Actual argument

- must have a shape
- can be an array section
- normally, a descriptor will be created and passed → no copying of data happens

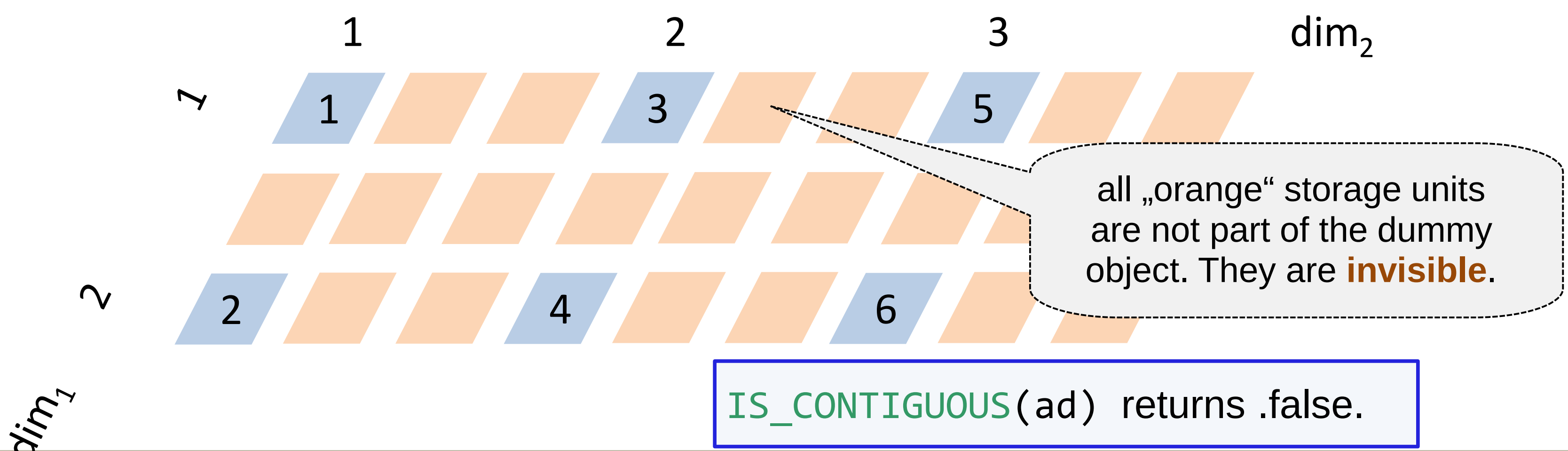
Memory layouts for assumed shape dummy objects

- **Actual argument is the complete array** `aa(0:1,3)`



`IS_CONTIGUOUS(ad)` returns .true.

- **Actual argument is an array section** `(0::2,1::3)` **of** `ab(0:2,9)`



`IS_CONTIGUOUS(ad)` returns .false.

F18

Note on assumed shape and interoperability

- **Example Fortran interface**

```
SUBROUTINE process_array(a) BIND(C)
  REAL(c_float) :: a(:,:)
END SUBROUTINE
```

assumed shape

- **Matching C prototype**

```
#include <ISO_Fortran_binding.h>

void process_array(CFI_cdesc_t *a);
```

Pointer to C descriptor

- **For an implementation in C, the header provides access to**
  - type definition of descriptor
  - macros for type codes, error states etc.
  - prototypes of library functions that generate or manipulate descriptors
- **Within a single C source file,**
  - binding is only possible to one given Fortran processor (no binary compatibility!)
- **Outside the scope of this course**

Internal procedures (1)

- **Example:**

```
SUBROUTINE process_expressions(…)
  REAL :: x1, x2, x3, x4, y1, y2, y3, y4, z
  REAL :: a, b
  a = …; b = …
  z = slin(x1, y1) / slin(x2, y2) + slin(x3, y3) / slin(x4, y4)
  …
CONTAINS
  REAL FUNCTION slin(x, y)
    REAL, INTENT(in) :: x, y
    slin = a * x + b * y
  END FUNCTION slin
  SUBROUTINE some_other(…)
    …
    … = slin(p, 2.0)
  END SUBROUTINE some_other
END SUBROUTINE process_expressions
```

host scoping unit (could be main program or any kind of procedure, except an internal procedure)

could be declared locally, or as dummy arguments

invocation within host

internal **function**

a, b accessed from the host → **host association**

internal **subroutine**

slin is accessed by **host association**

Internal procedures (2)

Rules for use

- invocation of an internal procedure is only possible inside the host, or inside other internal procedure of the same host
- an explicit interface is automatically created

Performance aspect

- if an internal procedure contains only a few executable statements, it can often be inlined by the compiler;
- this avoids the procedure call overhead and permits further optimizations

Legacy functionality: statement function OBS

```
SUBROUTINE process_expressions(…)
  REAL :: x, y
  slin(x, y) = a*x + b*y
  …
  z = slin(x1, y1) / slin(x2, y2) + slin(x3, y3) / slin(x4, y4)
END SUBROUTINE process_expressions
```

- should be avoided in new code

Controlling access to host

```
SUBROUTINE process_expressions(…)
  IMPLICIT NONE
  REAL :: x1, x2, x3, x4, y1, y2, y3, y4, z
  REAL :: a, b
  a = …; b = …
  z = slin(x1, y1) / slin(x2, y2) + slin(x3, y3) / slin(x4, y4)
  …
CONTAINS
  REAL FUNCTION slin(x, y)
    IMPORT, ONLY : a, b
    REAL, INTENT(in) :: x, y
    slin = a * x + b * y
  END FUNCTION slin
END SUBROUTINE process_expressions
```

- **Extension of the IMPORT statement**
  - assure that only specified objects from the host are visible
  - `IMPORT, NONE` blocks all host access
  - avoid unwanted side effects (both semantics and optimization) by enforcing the need to redeclare variables in internal procedure scope
- **Note: this is a F18 feature**
  - it is available in the most recent Intel compiler (19.0)

Subprograms with alternate returns

OBS

Purpose:

- permit subroutine to control execution of caller
- e.g., for error conditions
- (irregular) * form of dummy argument

```
SUBROUTINE gam(a, *, *)
  REAL :: a
  IF (a < -1.0) RETURN 1
  IF (a > 1.0)  RETURN 2
  a = SQRT(1-a*a)
  RETURN
END SUBROUTINE
```

Calling program unit

- actual arguments refer to **labels** defined in calling unit

```
   CALL gam(a, *7, *13)
   :                       normal termination
   STOP 'SUCCESS'
 7 WRITE(*,*) 'too small'
   STOP 'ERROR'
13 WRITE(*,*) 'too big'
   STOP 'ERROR'
```

Typical error handling scheme in procedure

- **Use an optional integer status argument**

```
SUBROUTINE gam(a, stat)
  REAL, INTENT(INOUT) :: a
  INTEGER, OPTIONAL, &
          INTENT(OUT) :: stat
  INTRINSIC :: SQRT
  INTEGER :: stloc
  stloc = 0
  IF (a < -1.0) THEN
    stloc = 1
  ELSE IF (a > 1.0) THEN
    stloc = 2
  ELSE
    a = SQRT(1-a*a)
  END IF
```

convention for "success" (annotation on `stloc = 0`)

```
  IF (PRESENT(stat)) THEN
    stat = stloc
  ELSE IF (stloc /= 0) THEN
    ERROR STOP 1
  END IF
END SUBROUTINE gam
```

obligatory (annotation on `IF (PRESENT(stat))`)

- **Notes**
  - F95: PRESENT intrinsic returns .TRUE. if an actual argument is associated with an OPTIONAL argument (explicit interface is needed)
  - F08: ERROR STOP causes error termination

Possible invocations - Style suggestion for error handling

Variant 1:

```
REAL :: x

x = 0.7
CALL gam(x)
x = 1.5
CALL gam(x)
```

- fine: produces 0.71414 (for the first `CALL gam(x)`)
- terminates during procedure execution (for the second `CALL gam(x)`)

Variant 2:

```
INTEGER :: stat
comp : BLOCK
  REAL :: x
  x = …
  CALL gam(x, stat)
  IF (stat /= 0) EXIT comp
  :
END BLOCK comp
SELECT CASE (stat)
CASE(0)
  STOP 0
DEFAULT
  WRITE(*,*) 'ERROR:', stat
  ERROR STOP 1
END SELECT
```

- can declare variables here (at `REAL :: x`)
- will never terminate (at `CALL gam(x, stat)`)
- F18 allows replacing "1" by "stat" (at `ERROR STOP 1`)

Notes

- Variant 2 uses a F08 BLOCK construct for processing (permits avoiding GO TO)
- error handling happens after that construct (rather unimaginatively in this example)

Character string dummy arguments

Assumed length string

```
SUBROUTINE pass_string(c)
  INTRINSIC :: LEN
  CHARACTER(LEN=*) :: c
  WRITE(*,*) LEN(c)
  WRITE(*,*) c
END SUBROUTINE
```

keyword spec can be omitted

- string length is passed **implicitly**

Usage:

```
INTRINSIC :: TRIM
CHARACTER(LEN=20) :: str

str = 'This is a string'
CALL pass_string(TRIM(str))
CALL pass_string(str(9:16))
```

- produces the output

```
16
This is a string
8
a string
```

F03

Handling of strings that interoperate with C

- **Remember: character length must be 1 for interoperability**
- **Example: C prototype**

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

'\0'-terminated character sequence

- **matching Fortran interface**
  - declares `c_char` entity as a rank 1 assumed size array

```
INTERFACE
  INTEGER(c_int) function atoi(in) BIND(C)
    USE, INTRINSIC :: iso_c_binding
    CHARACTER(c_char), DIMENSION(*) :: in
  END FUNCTION
END INTERFACE
```

F03

Handling of strings that interoperate with C

- **Invoked by**

```
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
```

C string needs terminator

  - **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**

Name	Value in C
c_null_char	'\0'
c_new_line	'\n'
c_carriage_return	'\r'

most relevant subset

Global variables

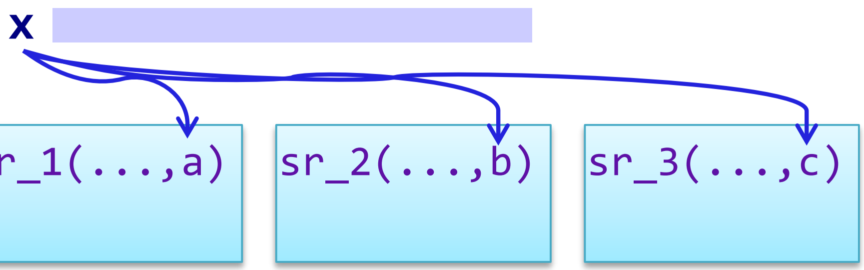
Global variables - Concept

Typical scenario:

- call **multiple** procedures which need to work on the **same** data

Well-known mechanism:

- data passed in/out as arguments

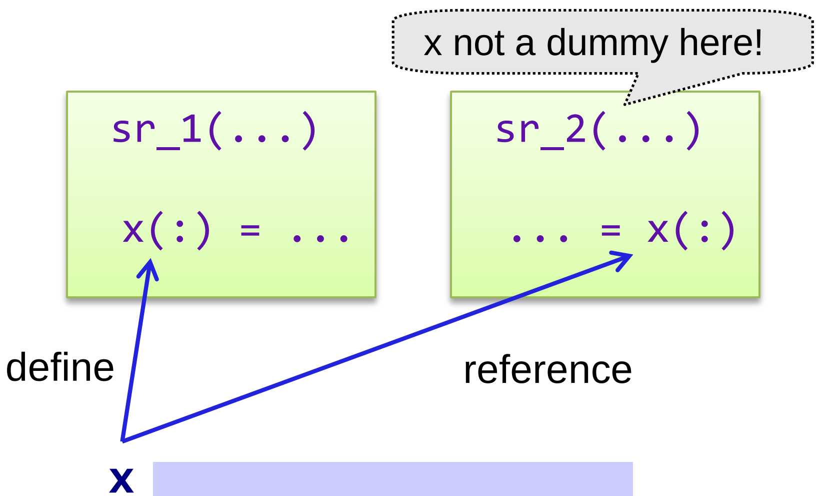


Consequences:

- need to declare in exactly one calling program unit → potential call stack issue
- access not needed from any other program unit (including the calling one)

Alternative:

- define **global storage area** for data
- accessible from subroutines without need for the invoker to provision it



- improvement of encapsulation

Fortran 77 style global data

OBS

- **COMMON block – a set of specification statements**
- **Example:**

required to be a sequence type

must be PARAMETERs

```
TYPE(field_type) :: f
REAL :: x(ndim), y(ndim, mdim)
INTEGER :: ic, jc, kc

COMMON / field_globals / ic, jc, kc, f, x, y
```

Name of block

Variable list

- typical best practice: put this in an include file `field_globals.finc` (note: this feature was not in Fortran 77, so a vendor extension)
- Usage in each procedure that needs access:

```
SUBROUTINE sr_1(...)
  INCLUDE "field_globals.finc"
  :
  x(:) = ...
  ...
```

modifies storage area represented by „x" inside the COMMON block

Semantics of COMMON

- **Block name is a global entity**
  - and therefore can be accessed from multiple program units;
  - it references a sequence of storage units.
  - Note: one unnamed COMMON block may exist.
- **List of variables**
  - of intrinsic type (or sequence type)
  - variables are „embedded“ into storage area in sequence of their appearance → determines size of storage area
  - number of storage units used for each variable: depends on its type
- **Why the „best practice“?**
  - avoid maintenance nightmare when changes are necessary
  - avoid confusion arising from
    (a) varying variable names
    (b) partial storage association
  - avoid ill-defined situations arising from type mismatches

Lifetime of data in COMMON

- **If a procedure that references `field_globals` completes execution**
  - and no other procedure that references it is active, the block becomes undefined
- **Prevent this undefinedness by adding**

```
SAVE / field_globals /
```

  to the include file

- **Definition status of objects in COMMON block**
  - may become defined after start of execution, or not at all

General problems with COMMON:

- Data flow is non-intuitive, especially if very many program units access the COMMON block.
- Negatively impacts comprehensibility and maintainability of code
- Many restrictions (e.g. no dynamic data) and limitations (e.g. type system)

Initialization of COMMON data

Special program unit

OBS

```
BLOCK DATA init_field_globals
   IMPLICIT NONE
   INCLUDE "field_globals.finc"
   DATA x / ndim * 1.0 /
   :
END BLOCK DATA
```

- uses a DATA statement to initialize some or all variables inside one or more named COMMON blocks
- Multiple BLOCK data units can exist, but they must avoid initializing the same block

Assure initialization

- is performed at program linkage time
  (Data vs BSS section of memory)

```
PROGRAM sim_field
  IMPLICIT NONE
  EXTERNAL :: init_field_globals
  ...
END PROGRAM
```

Unnamed BLOCK DATA

- one is possible, but then initialization requires a compiler switch for linkage.

Conversion to encapsulated module variables

```
MODULE mod_field_globals
  IMPLICIT NONE
  PRIVATE
  PUBLIC :: setup_field_globals, sr_1
  :
  REAL :: x(ndim), y(ndim,mdim)
  INTEGER :: ic, jc, kc
CONTAINS
  SUBROUTINE setup_field_globals(...)
    :
  END SUBROUTINE
  SUBROUTINE sr_1(...)
    ...
    x(:) = ...
  END SUBROUTINE sr_1
  ... ! sr_2, sr_3 etc.
END MODULE
```

USE access only to specified procedures

initialize x, y

has access to x by host association

Usage:

```
PROGRAM sim_field
  USE mod_field_globals
  IMPLICIT NONE
  :
  CALL setup_field_globals(...)
  :
  CALL sr_1(...)
  : ! call sr_2, sr_3 etc.
END PROGRAM
```

x, y, ... are inaccessible

Note:

- module variables and local variables of the main program always have the SAVE attribute

Initialization of module variables
(illustrative example)

```
MODULE mod_field_globals
  IMPLICIT NONE
  PRIVATE
  :
  PUBLIC :: f
  TYPE :: field_type
    PUBLIC
    REAL :: x(ndim), y(ndim)
  END TYPE
  TYPE(field_type) :: f = &
            field_type( x=[ (0.0,i=1,ndim) ], y=[ (0.0,i=1,ndim) ] )
  :
END MODULE
```

type is PRIVATE, but object is PUBLIC
→ f is the only object of that type

F03

initialization expression

```
PROGRAM sim_field
  USE mod_field_globals
  IMPLICIT NONE
  :
  ... = f%x(:)
END PROGRAM
```

access to use associated f permitted

- **Note:**
  - `TYPE(field_type)` need not be a sequence type here
  - Objects existing only once: **Singleton** pattern

Global data and interoperability

Defining C code:

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

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

Mapping Fortran code:

```
MODULE mod_globals
  USE, INTRINSIC :: iso_c_binding

  INTEGER(c_int), BIND(c) :: ic
  REAL(c_float) :: rpar(4)
  BIND(c, name='Rpar') :: rpar
end module
```

- either attribute or statement form may be used

- 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.
- BIND(C) COMMON blocks are permitted, but obsolescent.

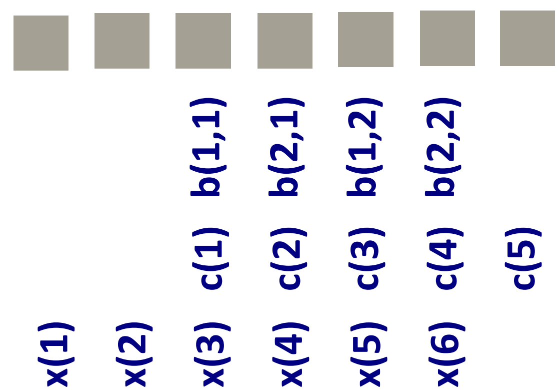
Enforcing storage association

- **EQUIVALENCE statement** OBS
  - use same memory area for two different objects
- **Example 1: Aliasing**

```
REAL :: x(6), b(2,2), c(5)
EQUIVALENCE (x(3), b, c)
```

same type for all objects

  - storage layout:



- **Example 2: Saving memory at cost of type safety**

need to avoid using undefined values
→ use in disjoint code sections

```
REAL    :: y(ndim)
INTEGER :: iy(ndim)
EQUIVALENCE (y, iy)
```

Replacement mechanisms

- **Example 1 from previous slide: Use pointers**

```
REAL, TARGET  :: x(6)
REAL, POINTER :: b(:2,:2) => x(2:),
                 c(:) => x(2:)
```

F08

- **Example 2 from previous slide:**
  - Use allocatable variables if memory really is an issue

```
REAL, ALLOCATABLE    ::  y(:)
INTEGER, ALLOCATABLE :: iy(:)
:
ALLOCATE(y(ndim))
:
DEALLOCATE(y)
ALLOCATE(iy(ndim))
:
DEALLOCATE(iy)
```

- **Representation change**
  - Use the TRANSFER intrinsic if really needed

Dynamic memory

Dynamic objects executive summary

- **Add a suitable attribute to an entity:**

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

initial state is „unallocated“

rank 1 array with deferred shape

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

initial state is "unassociated"

- **Typical life cycle management:**

use of heap memory

create:

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

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

use:

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

definitions and references

destroy:

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

- **Status checking:** **(hints at semantic differences!)**

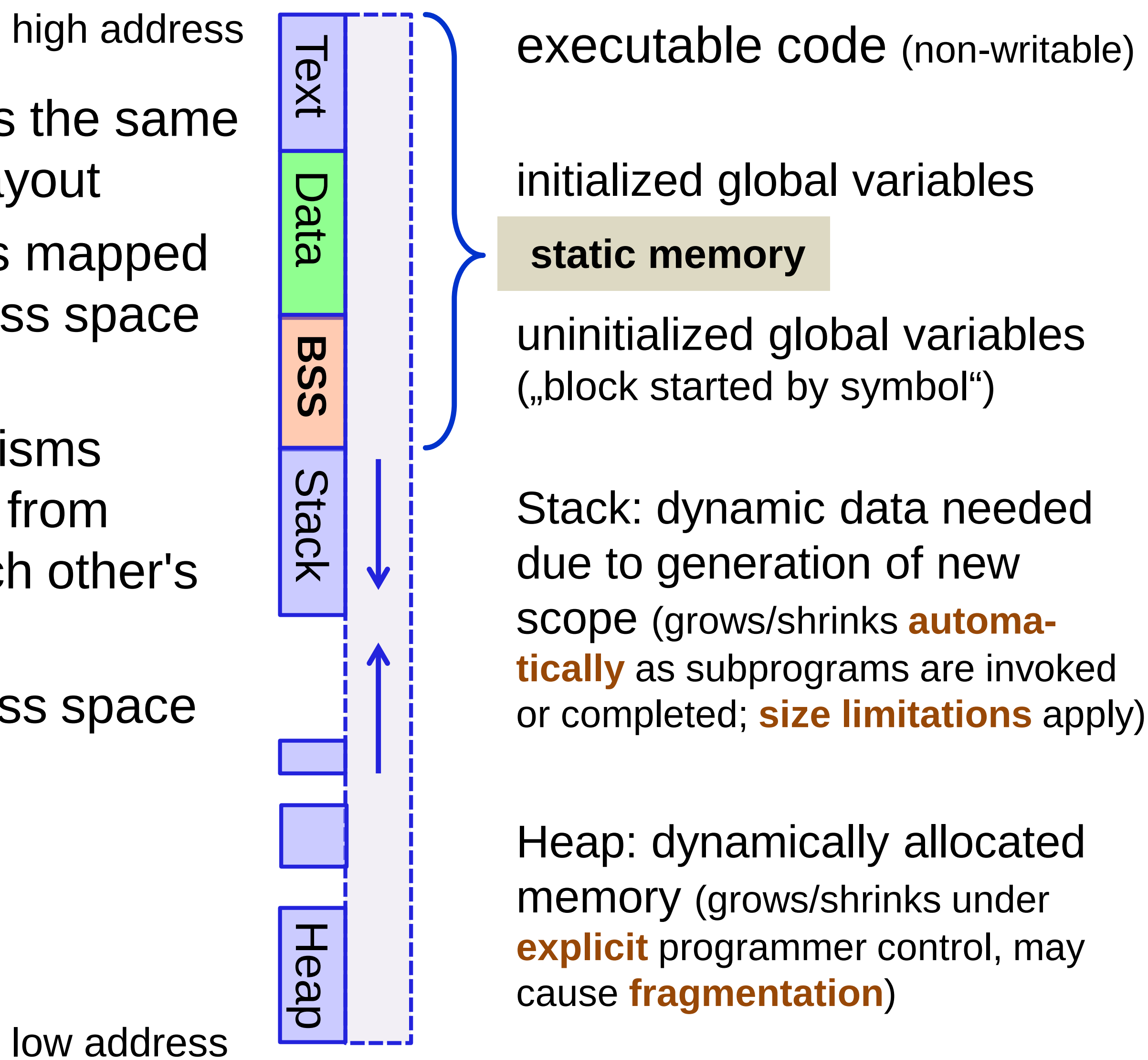
```
IF (ALLOCATED(x)) THEN; ...
```

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

logical functions

Some remarks about memory organization

- **Virtual memory**
  - every process uses the same (formal) memory layout
  - physical memory is mapped to the virtual address space by the OS
  - protection mechanisms prevent processes from interfering with each other's memory
  - 32 vs. 64 bit address space



ALLOCATABLE vs. POINTER

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

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

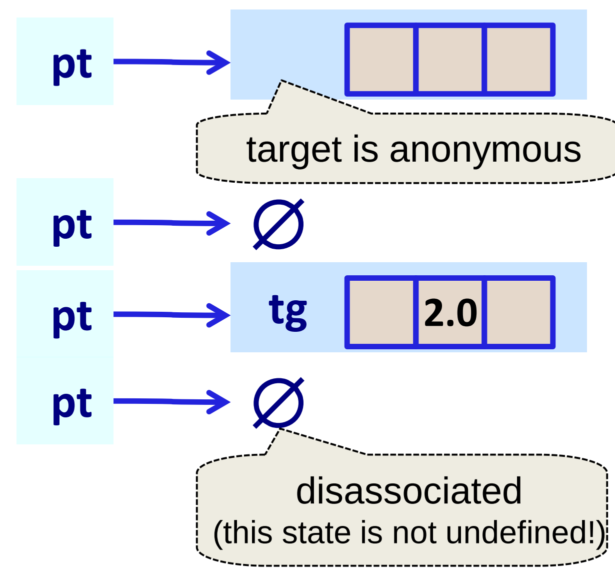
- **An associated pointer entity**
  - is an alias for another object, its target
  - all definitions and references are to the target

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

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

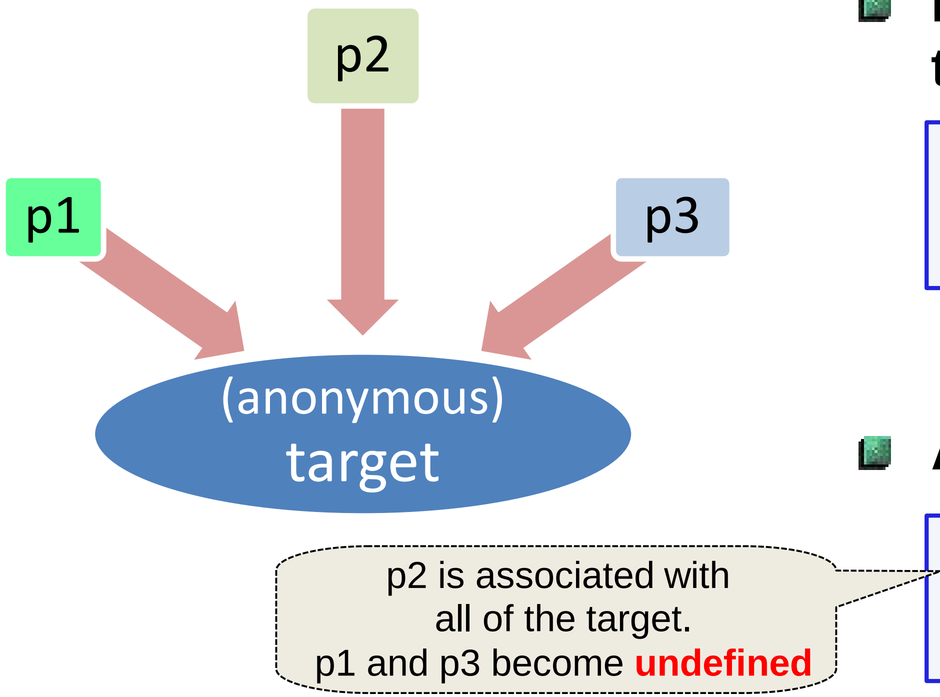
> essential, otherwise an orphaned target can remain

> explicit pointer assignment



- undefined (third) state should be avoided

Implications of POINTER aliasing



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

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

- **Avoid dangling pointers**

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

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

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

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

Features added in F03

Allocatable entities

- Scalars permitted:

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

- LHS auto-(re)allocation on assignment:

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

conformance LHS/RHS guaranteed, but: additional run-time check

- Efficient allocation move:

avoid data movement

```
CALL MOVE_ALLOC(from, to)
```

Pointer entities

- rank changing „=>“:

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

rank of target must be 1

- bounds changing „=>“:

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

bounds remapped via lower bounds spec

Deferred-length strings:

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

var_string = 'String of any length'
```

POINTER also permitted, but subsequent use is then different

LHS is (re)allocated to correct length

Special case: automatic variables

- **Run-time sizing of local variables**

```
MODULE mod_proc
  INTEGER, PARAMETER :: dm = 3, &
                        da = 12
CONTAINS
  SUBROUTINE proc(a, n)
    REAL, INTENT(inout) :: a(*)
    INTEGER, INTENT(in) :: n
    REAL :: wk1(int(log(real(n))/log(10.)))
    REAL :: wk2(sfun(n))
    :
  END SUBROUTINE proc
  PURE INTEGER function sfun(n)
    INTEGER, INTENT(in) :: n
    sfun = dm * n + da
  END FUNCTION sfun
END MODULE mod_proc
```

  - by use of specification expressions

- **A special-case variant of dynamic memory**
  - usually placed on the stack
- **An automatic variable is**
  - brought into existence on entry
  - deleted on exit from the procedure
- **Note:**
  - for many and/or large arrays creation may fail due to stack size limitations – processor dependent methods for dealing with this issue exist

Intel ifort: `-heap-arrays`

F03

ALLOCATABLE dummy argument
(explicit interface required)

- **Useful for implementation of „factory procedures“**
  - e.g., by reading data from a file

```
SUBROUTINE read_simulation_data( simulation_field, file_name )
  REAL, ALLOCATABLE, INTENT(out) :: simulation_field(:,:,:)
  CHARACTER(LEN=*), INTENT(in) :: file_name
  INTEGER :: iu, n1, n2, n3

  OPEN(NEWUNIT=iu, FILE=file_name, …)
  READ(iu) n1, n2, n3
  ALLOCATE( simulation_field(n1,n2,n3) )
  READ(iu) simulation_field
  CLOSE(iu)
END SUBROUTINE read_simulation_data
```

deferred-shape

storage can be allocated after determining its size

- **Actual argument**
  - that corresponds to `simulation_field` must be ALLOCATABLE (apart from having the same type, kind and rank)

F95

POINTER dummy argument
(explicit interface required)

POINTER dummy argument

- Example 1: for use as the RHS in a pointer assignment

```
SUBROUTINE process_as_target( …, item )
  REAL, POINTER, INTENT(in) :: item(:)
  IF (ASSOCIATED(item)) THEN
    some_pointer => item
    :
    some_pointer(j) = …
  END IF
END SUBROUTINE
```

deferred-shape (annotation on `item(:)`)

modifies target of `item` (annotation on `some_pointer(j) = …`)

- Example 2: for use as the LHS in a pointer assignment

```
SUBROUTINE process_as_pointer( …, item )
  REAL, POINTER, INTENT(inout) :: item(:)
  IF (.NOT. ASSOCIATED(item)) item => some_target(j,:)
  :
  item(k) = …  ! guarantee associatedness at this point
END SUBROUTINE process_as_pointer
```

deferred-shape (annotation on `item(:)`)

Invocation of procedures with POINTER dummy argument

Example 1:

```
REAL, POINTER :: p(:) => NULL()
REAL, TARGET :: t(ndim)

ALLOCATE( p(ndim) )
CALL process_as_target( …, p )

CALL process_as_target( …, t(::2) )
```

Auto-targetting (F08)
Permitted for INTENT(in) pointers

Example 2:

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

CALL process_as_pointer( …, p )
:
CALL process_as_pointer( …, p )
```

First call: unassociated on entry, set up object in procedure

Second call: associated on entry, continue working on same object

- here, the actual argument must have the POINTER attribute

INTENT semantics for dynamic objects

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

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

Bounds of deferred-shape objects

- **Bounds are preserved across procedure invocations and pointer assignments**
  - Example:

```
REAL, POINTER :: my_item(:) => NULL()

ALLOCATE(my_item(-3:8))
CALL process_as_target(…, my_item)
```

What arrives inside the procedure? Use intrinsics to check ...

```
SUBROUTINE process_as_target(…)
  :
  some_pointer%item => item
```

`LBOUND(item)` has the value [-3]
`UBOUND(item)` has the value [8]
same applies for LHS after pointer assignment

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

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
  - aliasing via pointers has additional negative performance impact
- **A few scenarios where pointers may not be avoidable:**
  - information structures → program these in an encapsulated manner (see later for how to do that): 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 (loss of spatial locality, supression of vectorization)!

Interoperation with C pointer types

Situations not yet covered:

- How to write a Fortran type declaration matching the C type

```
typedef struct vector {
  int len;
  float *f;
} Vector;
```

- How to write a Fortran interface matching the C prototypes

```
double fun(double x, void *)
```

or

```
float strtof(const char *nptr, char **endptr);
```

F03

Typeless C pointers in Fortran

- **Opaque derived type defined in ISO_C_BINDING:**
  - `c_ptr`: interoperates with a `void *` C object pointer
- **Useful named constant:**
  - `c_null_ptr`: C null pointer

```
TYPE(c_ptr) :: p = c_null_ptr
```

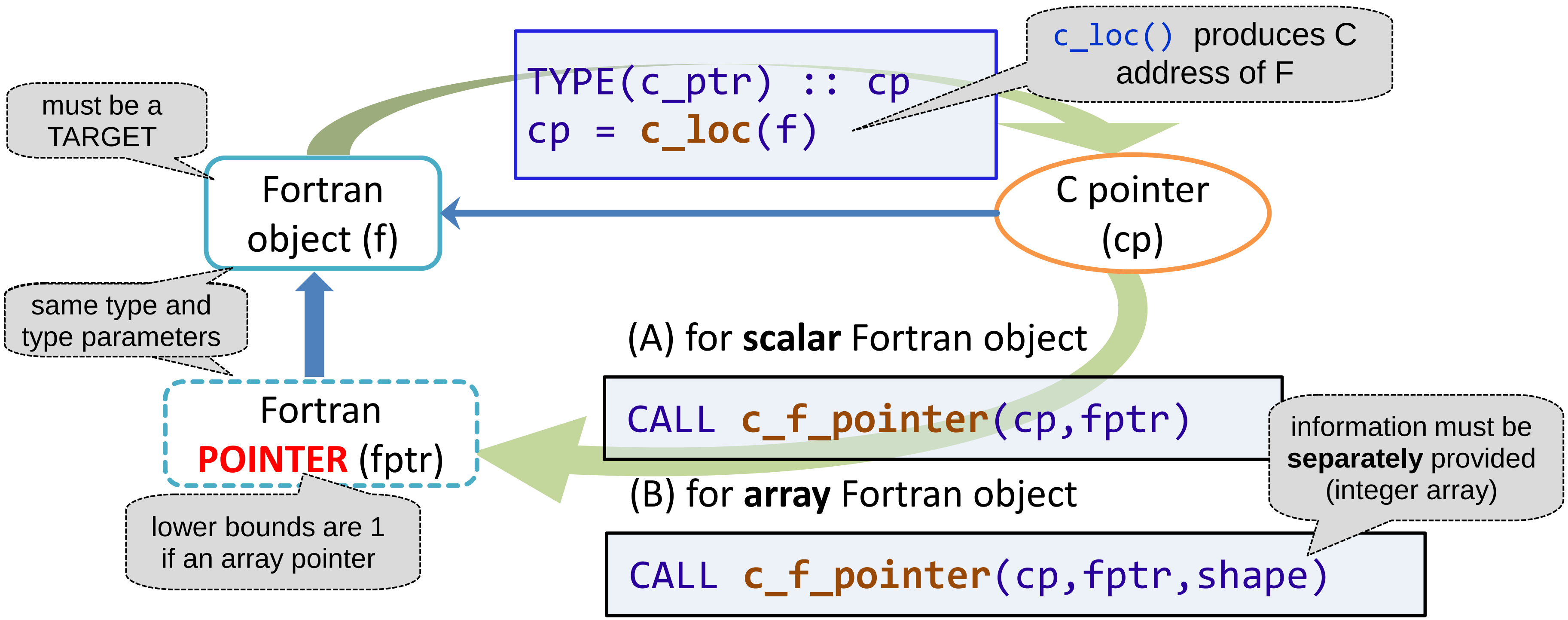
- **Logical module function that checks pointer association:**
  - `c_associated(c1[,c2])`
  - value is `.FALSE.` if `c1` is a C null pointer or if `c2` is present and points to a different address. Otherwise, `.TRUE.` is returned
  - typical usage:

```
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
```

F03

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)

Two scenarios are covered

1. **Fortran object is of interoperable type and type parameters**

2. **Fortran object is a non-interoperable variable**
   - non-polymorphic
   - no length type parameters

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

In both scenarios, the Fortran object must

- have either the POINTER or TARGET attribute
- be allocated/associated if it is ALLOCATABLE/POINTER
- be CONTIGUOUS and of non-zero size if it is an array

Note: some restrictions present in F03 were dropped in F18

Writing an interoperable Fortran type declaration

- **The following declarations are for interoperable types:**

```
typedef struct vector {
  int len;
  float *f;
} Vector;
```

↕

```
TYPE, BIND(C) :: vector
  INTEGER(c_int) :: len
  TYPE(c_ptr) :: f
END TYPE
```

  - note that type and component names need not be the same
- **Further details are left to the exercises**

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

Optional slide

Cray Pointers: pre- F95 dynamic memory management

- **Not part of any Fortran standard**
  - functionality first introduced by Cray as an extension
- **Declaration**

```
REAL :: arr(1)
POINTER (ptr, arr)
```

pointee

  - integer pointer `ptr` is (automatically) of an integer of a kind suitable for representing a C pointer (system-dependent!)
  - pointee: entity of any type (usually intrinsic or sequence), scalar or array
  - the POINTER, ALLOCATABLE or TARGET attributes are **not** permitted for the pointee

Optional slide

Memory management procedures

Dynamic allocation and deallocation

- uses non-standard intrinsics:

```
PTR = malloc(nsize * sz_real)
arr(1:nsize) = [ … ]
: ! further processing of arr
CALL free(ptr)
```

pointee now has new start address

- note that arguments are in units of bytes → you need to know sizes of storage units
- ⚠ for some compilers, `%val` must be used on the arguments of `malloc` and `free`
- ⚠ names and semantics of allocation and freeing procedures may differ between implementations
- data are accessed via pointee
- pointee array bounds checking will be suspended
- explicit deallocation is required to avoid memory leakage

Optional slide

Aliasing and pointer arithmetic

Arithmetic usually in units of bytes:

```
REAL :: x
POINTER(pnew, x(1))
:                           ! define ptr as in previous slide
pnew = ptr + sz_real * 2
WRITE(*,*) x(2)             ! value is that of arr(4)
```

- i.e., `x(:)` is aliased with `arr(3:)` via `pnew`
- some systems may use units of multi-byte words instead of bytes
- Incrementing `ptr` itself is possible, but may result in a memory leak

Performance impact

- will happen in the scope where the pointer is declared because of potential aliasing
- programmer's responsibility to avoid aliasing in other scopes!

Re-pointing a Cray pointer

Example: re-pointing at a global variable

```
MODULE mod_global
  DOUBLE PRECISION, SAVE :: arr_static(8) = [ … ]
END MODULE
PROGRAM global
  USE mod_global
  REAL :: arr(1)
  DOUBLE PRECISION :: darr(1)
  POINTER (ptr, arr), (ptr, darr)
  : ! use ptr via arr as shown previously

  ptr = loc(arr_static)
  WRITE(*,*) darr(2)
END PROGRAM
```

might be in a COMMON block if older code base is used

accesses to `arr` would produce undefined results

- multiple pointees of different type → use the correct one!
- `darr` is aliased with `arr_static` after execution of pseudo-intrinsic `loc`

Example code:
`examples/cray_ptr/cray_pointers.f90`

Optional slide

Additional notes

- **Some compilers require additional switches / libraries:**
  - gfortran: `-fcray-pointer`
  - xlf: `-qalias=intptr -qddim … -lhm`

  → please study your compiler documentation
- **Some compilers also support pointing at procedures**
  - not really portable – was not supported by original Cray concept
  - „real“ procedure pointers are supported in F03

Optional slide

Moving to standard-conforming code

Option 1: Use ALLOCATABLE entities

- this conversion is easy to do if only the dynamic memory facility (malloc/free) is used (no aliasing!)

Example code that nearly matches semantics:
`examples/cray_ptr/ftn_alloc.f90`

Option 2: Use POINTER entities

- this conversion is moderately easy to do; pointer arithmetic must be converted to pointer array indexing

Example code that nearly matches semantics:
`examples/cray_ptr/ftn_pointers.f90`

The above two use pure F03 and typically require larger-scale rewriting, even though not necessarily difficult.

Optional slide

Moving to standard-conforming code (cont'd)

- **Option 3: Use C interoperability from** F03
  - this conversion allows for a more direct mapping of existing source code
  - especially relevant if targeted compiler does not support Cray pointers
- **Use the `c_ptr` type from `iso_c_binding`**
  - an object of that type can be used in place of a Cray pointer

```
REAL :: arr(1)
POINTER (ptr, arr)
```

→

```
USE, INTRINSIC :: iso_c_binding
:
REAL, POINTER :: arr(:)
TYPE(c_ptr) :: ptr
```

at this point, no relationship exists yet between `ptr` and `arr`

Memory management

- **It is possible to make direct use of libc facilities**
  - Fortran interface declaration for required functions:

```
INTERFACE
  TYPE(c_ptr) FUNCTION malloc(size) BIND(C)
    IMPORT :: c_ptr, c_size_t
    INTEGER(c_size_t), VALUE :: size
  END FUNCTION
  SUBROUTINE free(ptr) BIND(C)
    IMPORT :: c_ptr
    TYPE(c_ptr), VALUE :: ptr
  END SUBROUTINE
END INTERFACE
```

  - With the declaration change from the previous slide, the statement

```
ptr = malloc(nsize*sz_real)
```

units are bytes here!

to allocate the needed memory can therefore be retained!

Optional slide

Mapping the C pointer to Fortran objects

Construct Fortran POINTER

- by using the intrinsic module procedure `c_f_pointer`,
- the memory part of which is identical with that pointed at by the `c_ptr` object

```
ptr = malloc(nsize*sz_real)
CALL c_f_pointer( ptr, arr, [nsize] )
```

rank-1 array pointer `arr` needs one upper bound

Re-pointing to a global variable

- Use `c_loc` to produce an address to be stored in a `c_ptr` object from a Fortran object (re-pointing scenario):

```
ptr = loc(arr_stat(4:))
```

Cray pointer

```
ptr = c_loc(arr_stat(4:))
```

C interop

The Fortran object is obliged to have the TARGET attribute, because `c_f_pointer` is likely to be subsequently applied to `ptr`

Optional slide

Wrapping up ...

- **Pointer arithmetic**
  - can be implemented with suitable operator overloading
- **Before C interop was available, Cray pointers were essential for some programming tasks**
  - e.g., use of the one-sided MPI calls

Example code that **fully** matches Cray pointer semantics:
`examples/cray_ptr/c_interop.f90`

Program configuration control

Metadata for program execution

Examples:

- Problem classes and sizes
- Parameter settings
- Names of input/output files

Small amounts of data!

- avoid encoding these into the program
- use dynamic allocation wrt problem sizes

Data format

- usually key-value pairs

Implementation methods

- environment variables
  → intrinsic procedure
  `GET_ENVIRONMENT_VARIABLE`
- command line arguments
  → intrinsic procedures exist
  → prefer to use a getopt-like abstraction layer
- NAMELIST files and variables
  → defined in the standard
- JSON
  → a language-independent API for structured processing of nested key-value pairs
  → Fortran implementation at https://github.com/jacobwilliams/json-fortran
  → Illustration of use at https://github.com/jacobwilliams/json-fortran/wiki/Example-Usage

Namelist processing (1)

- **Purpose:**
  - handling of key-value pairs
  - association of keys and values is defined in a file
  - a set of key value-pairs is assigned a name and called a namelist group
- **Example file:**

file my_nml.dat

```
&groceries flour=0.2,
    breadcrumbs=0.3, salt=0.01 /
&fruit apples=4, pears=1,
    apples=7 /
```

final value relevant

  - contains two namelist groups
  - first non-blank item: &
  - terminated by slash

- **Required specifications**

```
REAL :: flour, breadcrumbs, &
        salt, pepper
INTEGER :: apples, pears
NAMELIST /groceries/ flour, &
        breadcrumbs, salt, pepper
NAMELIST / fruit / pears, apples
```

- **Reading the namelist**

```
OPEN(12, FILE='my_nml.dat', &
     FORM='formatted', ACTION='read')
READ(12, NML=groceries)
! pepper is undefined
READ(12, NML=fruit)
```

  - **NML** specifier **instead** of **FMT**
  - multiple namelists require **same order** of reading as specified in file

Namelist processing (2)

- **Arrays**
  - namelist file can contain array values in a manner similar to list-directed input
  - declaration may be longer (but not shorter) than input list – remaining values are undefined on input
  - I/O is performed in array element order
- **Strings**
  - output requires DELIM specification

```
CHARACTER(LEN=80) :: name
NAMELIST /pers_nm/ name
name='John Smith'
OPEN(17, DELIM='quote', …)
WRITE(17, NML=pers_nm)
```

  otherwise not reusable for name-list input in case blanks inside string („too many items in input“)
  - input requires quotes or apostrophes around strings
- **Derived types**
  - form of namelist file (output):

```
&PERSON
ME%AGE=45,
ME%NAME=“R. Bader“,
YOU%AGE=33,
YOU%NAME=“F. Smith“
/
```

all Fortran objects must support the specified type components

- **Output**
  - generally uses large caps for identifiers

Command line processing via a user-friendly wrapper

FTN_Getopt

- module for handling command arguments of intrinsic type
- supported specifications are:

`--switch`	for a logical option (has value `.TRUE.` if option appears)
`--switch <value>`	or
`--switch=<value>`	for an otherwise typed option

Sequence of processing

1. invoke `optinit()` to create one or more options (scalar or array of type `opt_t`)
2. invoke `optarg()` to extract the option(s) from the command line
3. invoke `optval()` to obtain the result object

Example:

```
USE ftn_getopt
TYPE(opt_t) :: option
INTEGER :: nopt
option = optinit('nopt', 'integer')
CALL optarg(option)
CALL optval(option, nopt)
```

- last statement will transfer the value 42 to nopt if the program is invoked with the argument `--nopt 42`
- nopt will remain unchanged if no such option is encountered

Error handling

- type mismatches etc. cause abort unless `stat` arguments are specified

see https://www.lrz.de/services/software/programmierung/fortran90/courses/basic/doc_ftn_getopt/index.html

The Environment Problem

Unsolved problems with global variables

- **Problems appear in the context of parallel programming**
  - especially shared memory parallelism (OpenMP)
- **Variant 1:**
  - global variable needs to exist once for all thread context (a shared variable)
  - then, **all** updates and references must be via mutual exclusion (atomic, critical, or by locking/unlocking)
- **Variant 2:**
  - global variables exist, but need to be multiplexed to have one instance per thread context (threadprivate variables)
  - an elaborate scenario is supplied on the following slides
- **Both cases**
  - involve additional programming complexity

Setting the stage

- **Calculation of**

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

  where
  - *f(x,p)* is a real-valued function of a real variable x and a variable p of some undetermined type
  - a, b are real values
- **Tasks to be done:**
  - procedure with algorithm for establishing the integral → depends on the properties of *f(x,p)* (does it have singularities? etc.)

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

  - function that evaluates *f(x,p)*
- **Case study provides a simple example of very common programming tasks with similar structure in scientific computing.**

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

- **Interface:**

requested precision

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

uses a function argument

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

(user-provided function)

- **Invocation:**

define a, b

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

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

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

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

Mismatch of user procedure implementation

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

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

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

- **or like this**

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

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

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

Solution 1: Wrapper with global variables

```
MODULE mod_user_fun
  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
```

global variables (implies SAVE attribute)

has suitable interface for use with `d01ahf`

further procedures, e.g. `user_proc` itself

- **Usage:**

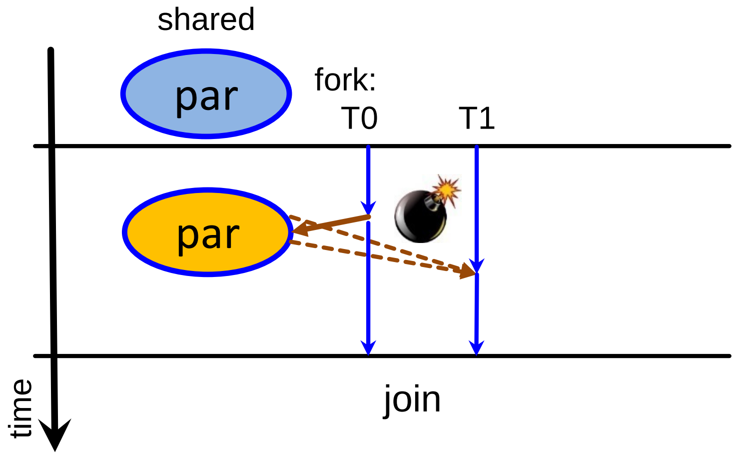
```
USE mod_user_fun

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

supply values for global variables

Disadvantages of Solution 1

- **Additional function call overhead**
  - is usually not a big issue (nowaday's implementations are quite efficient, especially if no stack-resident variables must be created).
- **Solution is 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
```



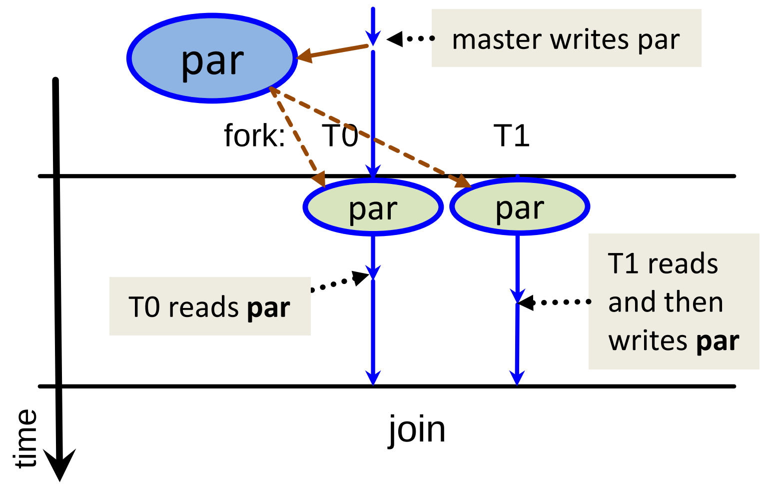
  - results in unsynchronized access to the **shared** variables `par` and `n` from **different** threads → race condition → does not conform to the OpenMP standard → **wrong results** (at least some of the time ...)

Making Solution 1 thread-safe

- **Threadprivate storage**

```
MODULE mod_user_fun
  DOUBLE PRECISION :: par
  INTEGER :: n
!$omp threadprivate (par, n)
   …
```

thread-individual copies are created in parallel regions



- **Usage may require additional care as well**

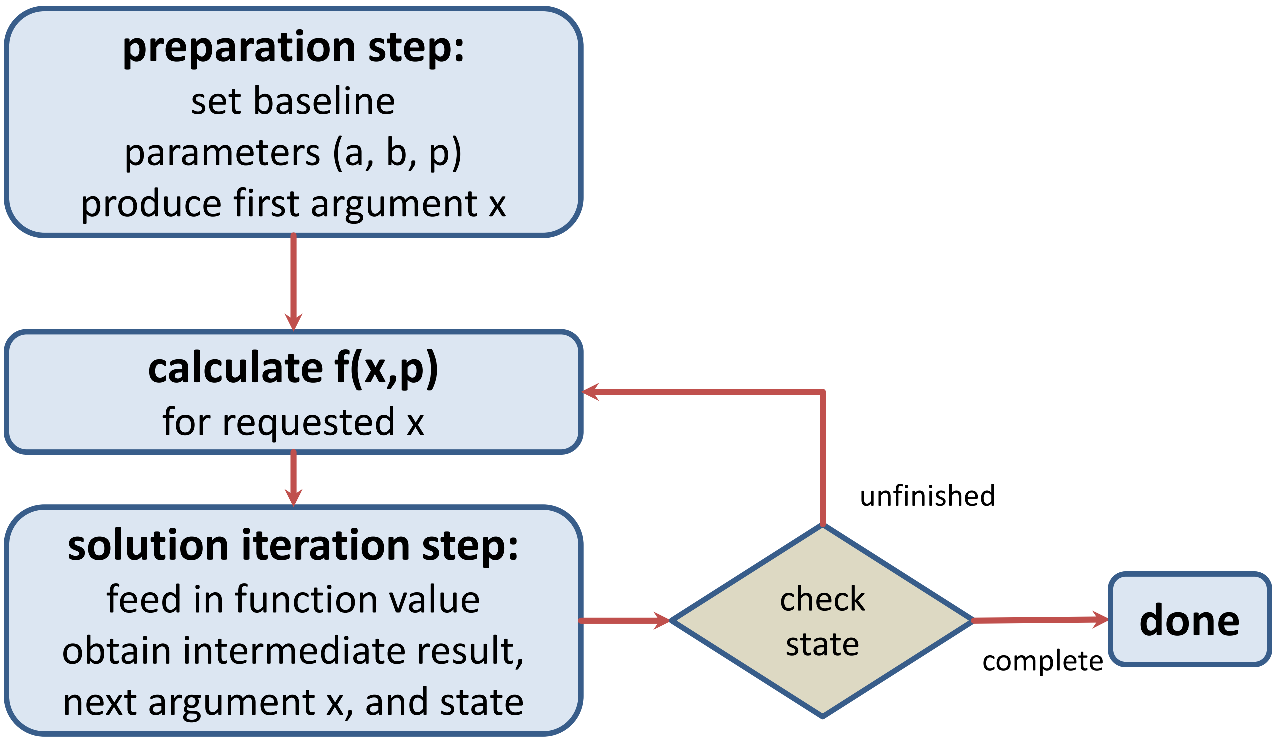
```
  par = …
!$omp parallel do copyin(par)
  DO i = istart, iend
    n = …
    … = d01ahf(…, arg_fun, …)
    IF (…) par = …
  END DO
!$omp end parallel do
```

broadcast from master copy needed for par

A bit cumbersome: non-local programming style required

Solution 2: Reverse communication

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



Solution 2: Typical example interface

- **Uses two routines:**

```
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
```

`result` shall not be modified by caller while calculation iterates

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

Solution 2: Using the reverse communication interface

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

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

Optional slide

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

Optional slide

Coroutine emulation via OpenMP tasking

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

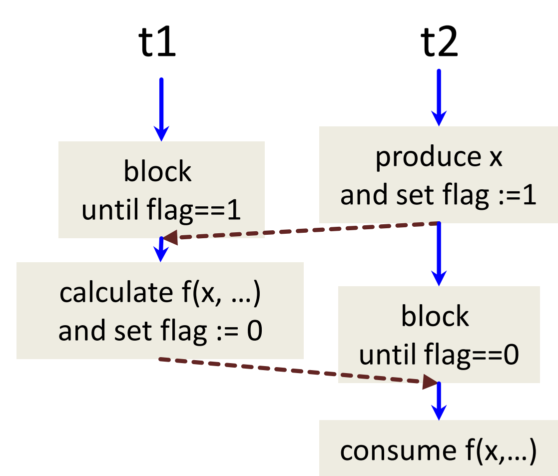
```
  :
  n = …; par = …; a = …; b = …; eps = …
  flag = flag_need_iter
!$omp parallel num_threads(2) proc_bind(master)
!$omp single
!$omp task …
    DO
      :
      CALL user_proc(x, n, par, fval)
      :
    END DO
!$omp end task
!$omp task
    CALL integrate_c(a, b, eps, fval, x, &
                     result, flag)
!$omp end task
!$omp end single
!$omp end parallel
  :
```

task t1

task t2

continues executing until the algorithm has completed

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



Optional slide

Synchronization code

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

```
!$omp task private(flag_local)
!$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

Solution 3: Object oriented design

- **Assume that parameter p in f(x, p) is passed to integration routine**

```
SUBROUTINE integrate_o(a, b, eps, fun, p, result, stat)
   :
   :
        … = fun(x, p)
   :
END SUBROUTINE
```

declarations given later

function dummy argument

parameters

- **Observation:**
  - integrator never makes explicit use of `p`
  - it is only passed to the invocation of the function argument
- **Idea:**
  - `p` should be a handle that can hold any data
  - we need to have a mechanism for accessing data implemented inside the procedure used as actual argument and associated with `fun`

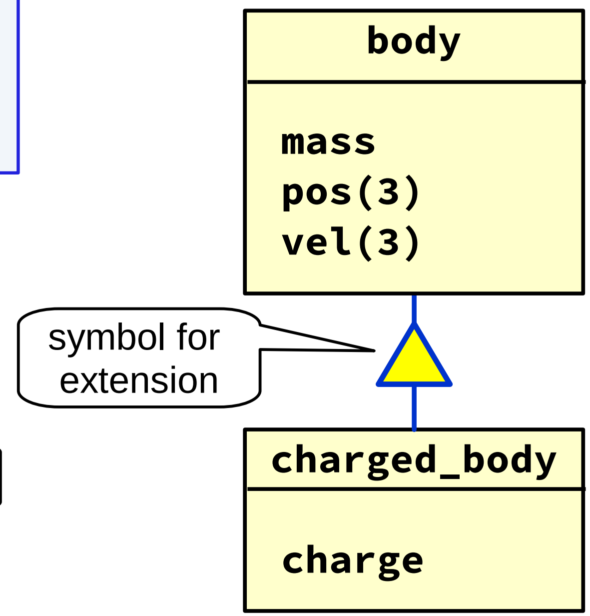
Object oriented features and their use

F03 Type extension

- **Example:**

```
TYPE, EXTENDS(body) :: charged_body
  REAL :: charge
END TYPE charged_body
```

- **UML-like representation**



- **Inheritance mechanism:**

```
TYPE(charged_body) :: electron
:
electron % mass = ...
electron % charge = ...
:
WRITE(*,*) electron % body
```

inherited component (electron % mass)

parent component (electron % body)

  - single inheritance only

Polymorphism

- **New capability of an object:**
  - permit change of type at run time

```
CLASS(body) :: particle
```

  - **declared** type is body
  - **dynamic** type can be declared type or any extension of it

- **Data item can be**
  1. a dummy data object
     interface polymorphism
  2. a pointer or allocatable variable
     data polymorphism →
     a new kind of dynamic memory
  3. both of the above

- **Properties of object:**
  - access to its data is by default possible only to components in declared type
  - an object of base type is type compatible with an object of extended type (but not vice versa)

Abstract type

Separate concerns in our integration example:

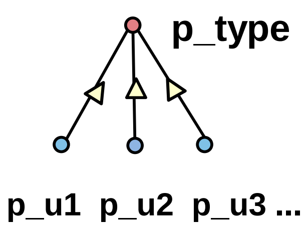
```
MODULE mod_integration
  :
  TYPE, ABSTRACT :: p_type
  END TYPE
  :
END MODULE mod_integration
```

```
MODULE mod_u1
  USE mod_integration
  :
  TYPE, EXTENDS(p_type) :: p_u1
    INTEGER :: n
    REAL(dk) :: par
  END TYPE
  :
END MODULE mod_u1
```

framework component

elaborated details

- no actual object of the abstract type can exist
  (even though type components are permitted)
- typical inheritance structure: flat tree



Completing the integrator framework

```
MODULE mod_integration
  :
  ABSTRACT INTERFACE
    FUNCTION fp(x, p) RESULT(r)
      IMPORT :: dk, p_type
      REAL(dk), INTENT(in) :: x
      CLASS(p_type), INTENT(in) :: p
      REAL(dk) :: r
    END FUNCTION fp
  END INTERFACE
CONTAINS
   SUBROUTINE integrate_o(a, b, eps, fun, p, result, stat)
    REAL(dk), INTENT(in) :: a, b, eps
    PROCEDURE(fp) :: fun
    CLASS(p_type), INTENT(in) :: p
    REAL(dk), INTENT(out) :: result
    INTEGER, INTENT(inout), OPTIONAL :: stat
    :
       … = fun(x, p)
    :
  END SUBROUTINE
END MODULE mod_integration
```

Describes signature of a function that is not yet implemented (ABSTRACT INTERFACE)

Enable access to host (IMPORT) F03

must be polymorphic, because **p_type** is abstract (CLASS(p_type) :: p)

Accessing data via type identification

```
MODULE mod_u1
  :
CONTAINS
  FUNCTION u1_fun(x, p) RESULT(r)
    REAL(dk), INTENT(in) :: x
    CLASS(p_type), INTENT(in) :: p
    REAL(dk) :: r

    SELECT TYPE (p)
    TYPE IS (p_u1)
      r = p%par * cos(p%n * x)
    CLASS default
      stop 'u1_fun: wrong type.'
    END SELECT
  END FUNCTION u1_fun
END MODULE mod_u1
```

Callout on the function header: must exactly match abstract interface (future use as actual argument!)

Callout on CLASS default: error handling

- **Specific integrand function implementation**
  - needs access to data stored in parameter object
- **SELECT TYPE**
  - F03 block construct
  - at most one block is executed
  - permits run time type identification (RTTI)
  - inside a `TYPE IS` guard, object is non-polymorphic and of the type specified in the guard
  - `CLASS IS` guards are also possible ("lift" declared type of a polymorphic object)

Calling the integrator

- **Main program**

```
PROGRAM integration
  USE mod_u1
  IMPLICIT NONE

  TYPE(p_u1) :: p
  REAL(dk) :: a, b, eps, …
  p%n = 4
  p%par = 3.4_dk
  a = …; b = …; …

  CALL integrate_o(a, b, eps, u1_fun, p, result, stat)

  WRITE(*,*) 'Result of integration: ', result
END PROGRAM integration
```

Acceptable as actual argument matching `class(p_type)` dummy because `p_type` is type compatible with `p_u1`

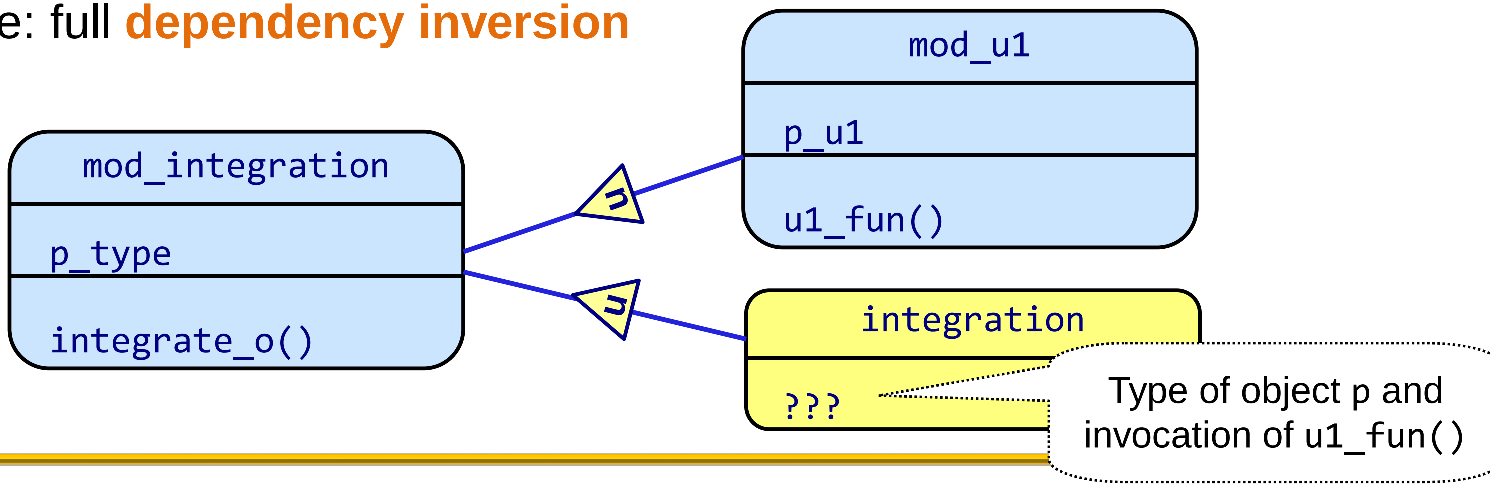
Remaining weaknesses

- **Weak spot 1: RTTI**
  - witness the need to do error handling in `u1_fun`
  - would be avoided if the argument could be declared

```
CLASS(p_u1), INTENT(in) :: p
```

  which is however not possible due to interface consistency constraints

- **Weak spot 2: Dependency tree of program units**
  - main program depends on specific implementation of type extension
    → needs rewrite+recompile to use a different parametrization scheme
  - desirable: full **dependency inversion**



F03

Binding procedures to types

Example:

- bind the `kick()` procedure to the type `body`

```
MODULE mod_body
 :
 TYPE :: body
   :
 CONTAINS
   PROCEDURE, PASS(this) :: kick
 END TYPE body
CONTAINS
 SUBROUTINE kick(this, dp)
   CLASS(body) :: this
   :
END MODULE
```

must be polymorphic

Invocation:

- through object

```
TYPE(body) :: particle
:
CALL particle % kick ( dp )
```

same as `call kick(particle, dp)`

- argument the object is passed at depends on PASS specification
- default is first one
- NOPASS: object is not passed
- only really interesting if actual argument is polymorphic

F03 Overriding type-bound procedures

Deferred type-bound procedure

```
MODULE mod_integration
  :
  TYPE, ABSTRACT :: p_type
  CONTAINS
    PROCEDURE(fp), PASS(p), &
                   DEFERRED :: fun
  END TYPE
  :
END MODULE mod_integration
```

- purpose is to force all type extensions to define an overriding type-bound procedure (and inform objects of declared base type that it exists)
- the existing abstract interface **fp** is referenced

Override for type extension

```
MODULE mod_u1
  USE mod_integration
  :
  TYPE, EXTENDS(p_type) :: p_u1
    INTEGER :: n
    REAL(dk) :: par
  CONTAINS
    PROCEDURE, PASS(p) :: &
                fun => u1_fun
  END TYPE
  :
END MODULE mod_u1
```

- signature of overriding procedure must be identical with that of **fp**, except for passed object

Implementation and invocation

Changes to u1_fun()

```
FUNCTION u1_fun(x, p) RESULT(r)
  REAL(dk), INTENT(in) :: x
  CLASS(p_u1), INTENT(in) :: p
  REAL(dk) :: r

  r = p%par * cos(p%n * x)
END FUNCTION u1_fun
```

- must replace `CLASS(p_type)` by `CLASS(p_u1)`
- RTTI not needed any more!
- implements **dynamic** dispatch (OO terminology: a virtual method)

Changes to integrator

- Function argument can be removed because function is now bound to the type

```
SUBROUTINE integrate_o(a, b, eps, &
                       p, result, stat)
    REAL(dk), INTENT(in) :: a, b, eps
    CLASS(p_type) :: p
    REAL(dk), INTENT(out) :: result
    INTEGER, INTENT(inout), &
                   OPTIONAL :: stat
    :
      … = p % fun(x)
    :
  END SUBRO
```

invoked function is the one bound to the **dynamic** type of p

Weakness 1 is hereby resolved

Addendum: binding procedures to objects

Use a procedure pointer

- declaration as type component

```
MODULE mod_body
  :
  TYPE :: body
    PROCEDURE(pr), POINTER :: &
                        print => null
  CONTAINS
    :
  END TYPE body
CONTAINS
  SUBROUTINE print_fmt(this)
    CLASS(body) :: this
    :
END MODULE
```

- **pr** references an abstract interface or an existing procedure

Example usage

- select print method for each object individually

```
TYPE(body) :: a, b

a%print => print_fmt
b%print => print_bin

CALL a%print() ! calls print_fmt
CALL b%print() ! calls print_bin
```

- invocation requires pointer components to be associated
- **PASS** attribute works as for type-bound procedures

Adressing weakness 2

- **Returning to the main program**

```
PROGRAM integration
  USE mod_integration

  CLASS(p_type), ALLOCATABLE :: p
  :
  fname = 'integration.dat'
  CALL p_class_create(p, fname)
  :
  CALL integrate_o(a, b, eps, p, result, stat)

  WRITE(*,*) 'Result of integration: ', result
END PROGRAM integration
```

type `p_u1` is not available via `mod_integrator` ...

this needs to return an allocated **p** of extended type

- here, the dependency structure is now OK, but the devil is in the details ...

Polymorphic factory method

F03

- **Uses sourced allocation to construct object**

```
:
  USE mod_u1
CONTAINS
  SUBROUTINE p_class_create (p, fname)
    CLASS(p_type), ALLOCATABLE, INTENT(out) :: p
    CHARACTER(len=*), INTENT(in) :: fname
    CHARACTER(len=strmx) :: type_string
    : ! open fname and read type_string
    SELECT CASE (type_string)
    CASE('p_u1')
      READ(…) n, par
      ALLOCATE( p, SOURCE=p_u1 (n, par) )
    CASE default
      ERROR STOP 'type not supported'
    END SELECT
  END SUBROUTINE
:
```

access definition of p_u1

use structure constructor for p_u1 to create a clone stored in p

Addendum: additional allocation mechanisms

F03

Typed allocation

```
CLASS(p_type), ALLOCATABLE :: p
:
ALLOCATE( p_u1 :: p )
```

- allocate **p** to be of dynamic type **p_u1**, but no value is provided

F08

Molded allocation

```
TYPE(p_u2) :: q
CLASS(p_type), ALLOCATABLE :: p
:
ALLOCATE( p, MOLD=q )
```

- allocate **p** to be of dynamic type **p_u2** (assuming **p_u2** is an extension of **p_type**), but does not copy over the value

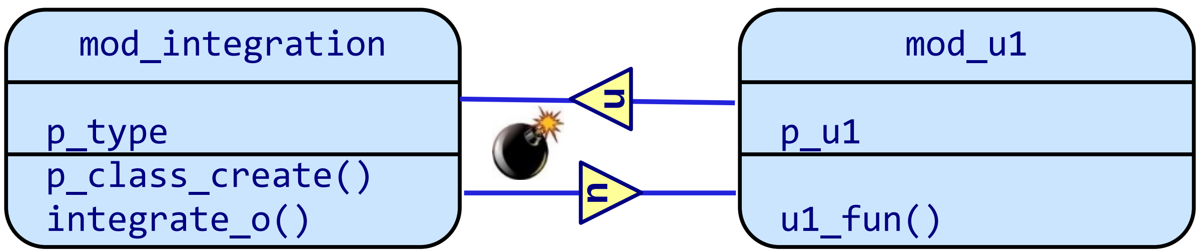
F08

Note:

- sourced and molded allocation also transfer array bounds

... and here's the catch

- **We can't have `p_class_create()` as a module procedure in `mod_integration`**
  - because this would create a circular module dependency:



- **On the other hand,**
  - its interface must be accessible via `mod_integration`

- **However,**
  - the interface's **signature** does not depend on `mod_u1`, ...

Submodules

A new kind of program unit

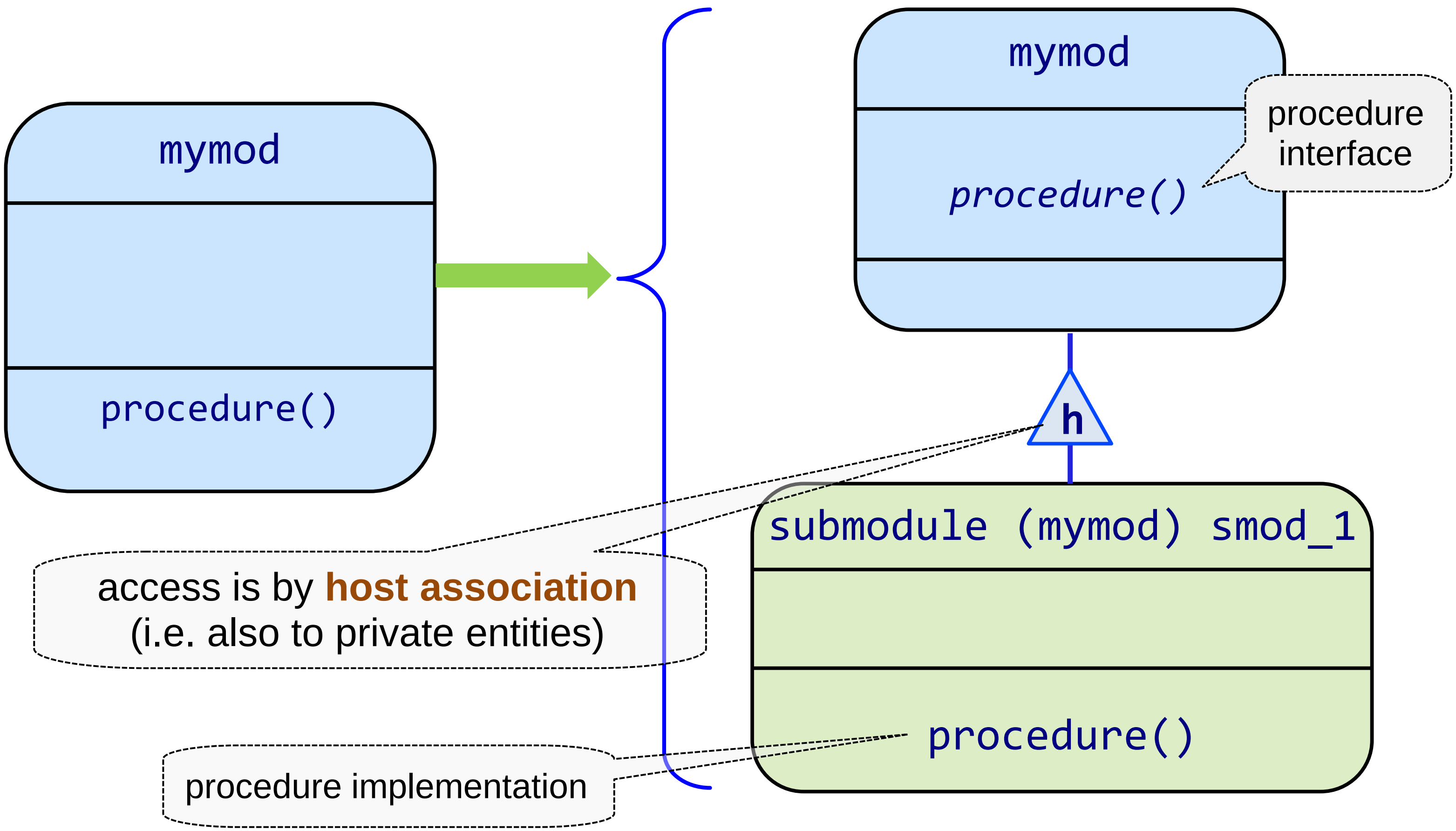
Problems with Modules

- **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
  - type definitions referencing each other may also occur in object-based programming

Solution: Submodules

F08

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



F08

Submodule program units

Syntax

ancestor module

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

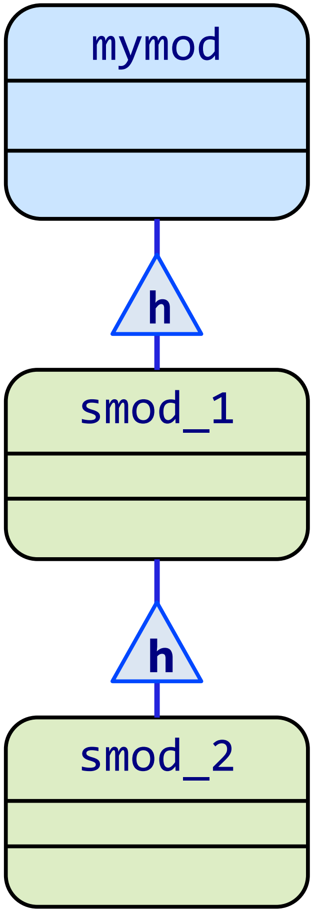
- applies recursively: a descendant of **smod_1** is

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

immediate ancestor submodule

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

Symbolic representation



F08

Submodule specification part

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

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

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

effectively private

F08

Separate module procedure interface

Returning to our integration example:

- specification part of ancestor module `mod_integration`

```
MODULE mod_integration
  :
  INTERFACE
    MODULE SUBROUTINE p_class_create (p, fname)
      CLASS(p_type), ALLOCATABLE, INTENT(out) :: p
      CHARACTER(len=*), INTENT(in) :: fname
    END SUBROUTINE
  END INTERFACE
END MODULE
```

syntax indication that the implementation is contained in a submodule

Notes:

- the `IMPORT` statement is not permitted in separate module procedure interfaces (auto-import is done)
- for functions, the syntax is `MODULE FUNCTION`

F08

Separate module procedure implementation

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

```
SUBMODULE (mod_integration) create
  USE mod_u1, ONLY : p_u1
CONTAINS
  MODULE PROCEDURE p_class_create
    CHARACTER(len=strmx) :: type_string
    : ! open fname and read type_string
    SELECT CASE (type_string)
    CASE('p_u1')
      READ(…) n, par
      ALLOCATE(p, SOURCE=p_u1(n, par))
    CASE default
      ERROR STOP 'type not supported'
    END SELECT
  END PROCEDURE
END SUBMODULE
```

Separate module procedure implementation

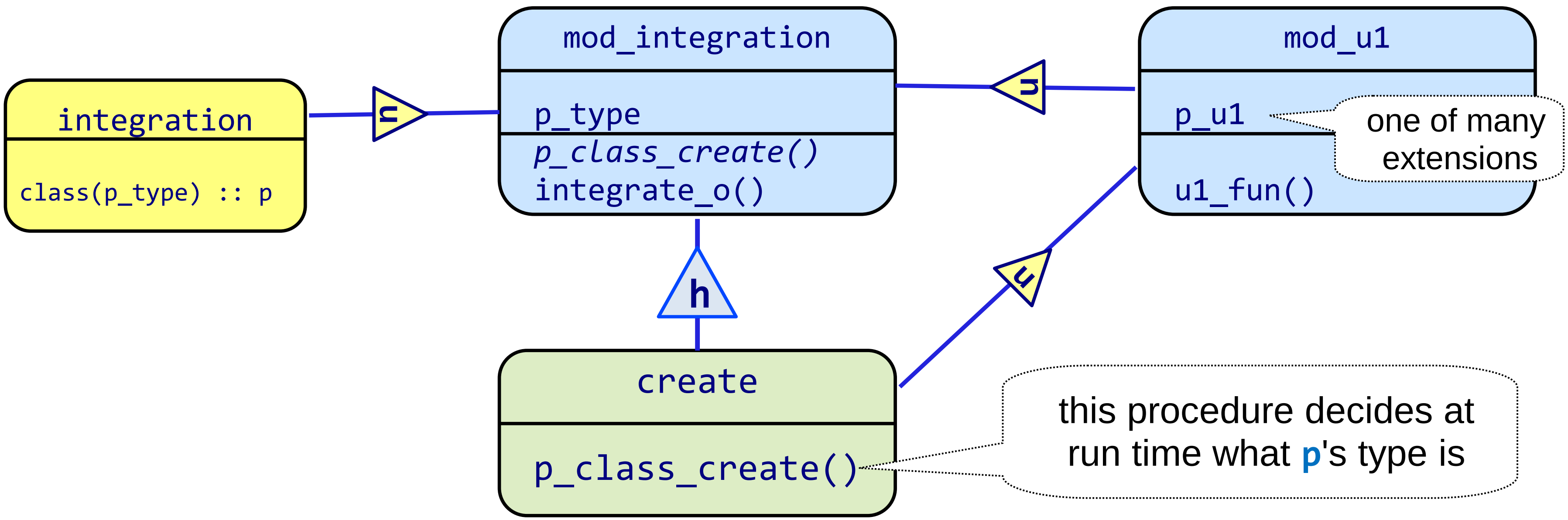
- **Syntax variant 2:**
  - interface is replicated in the submodule
  - must be consistent with ancestor specification

```
SUBMODULE (mod_integration) create
  USE mod_u1, ONLY : p_u1
CONTAINS
  MODULE SUBROUTINE p_class_create(p, fname)
    CLASS(p_type), ALLOCATABLE, INTENT(out) :: p
    CHARACTER, INTENT(in) :: fname
    : ! implementation as on previous slide
  END PROCEDURE
END SUBMODULE
```

  - for functions, the syntax again is `MODULE FUNCTION`

Weakness 2 is hereby resolved

Final dependency structure

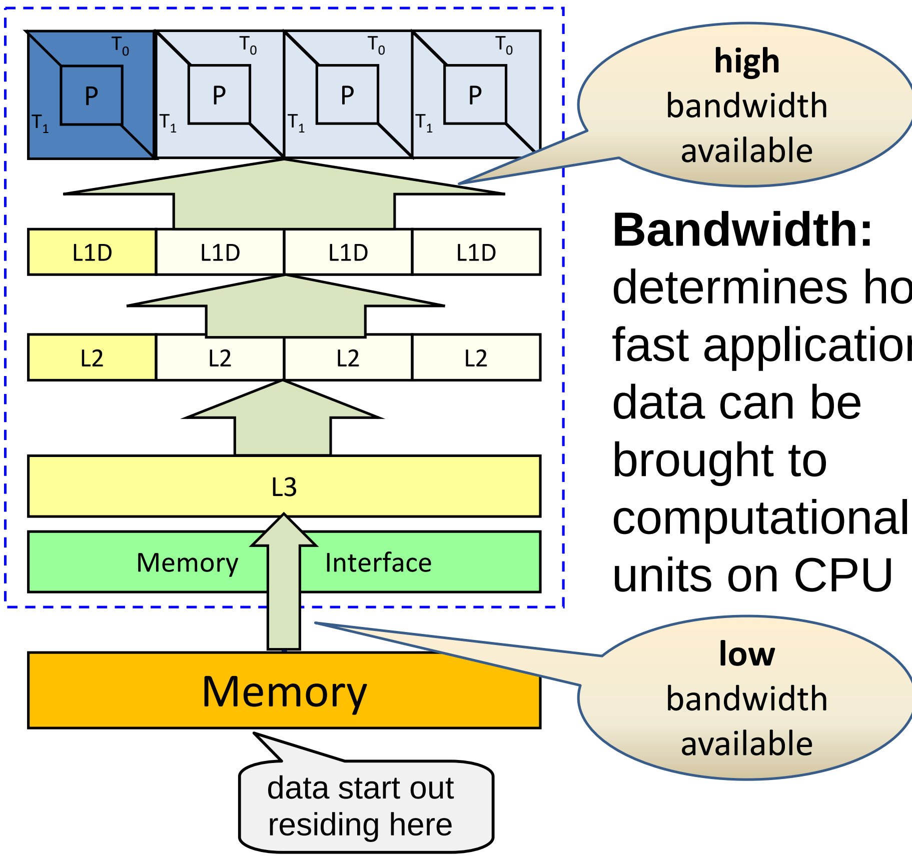


- **Notes:**
  - the standard permits use access (which usually is indirect) from a submodule to its ancestor module
  - since use association overrides host association, putting an **ONLY** option on **USE** statements inside submodules is recommended to avoid side effects resulting from encapsulation

Array Processing and its performance

Processor architecture

- **Performance Characteristics**
  - determined by memory hierarchy



Bandwidth: determines how fast application data can be brought to computational units on CPU

- **Impact on Application performance: depends on where data are located**
  - **temporal locality:** reuse of data stored in cache allows higher performance
  - **no temporal locality:** reloading data from memory (or high level cache) reduces performance
- **For multi-core CPUs,**
  - available bandwidth may need to be shared between multiple cores

  → shared caches and memory

Characteristics

- known operation count, load/store count
- some variants of interest:

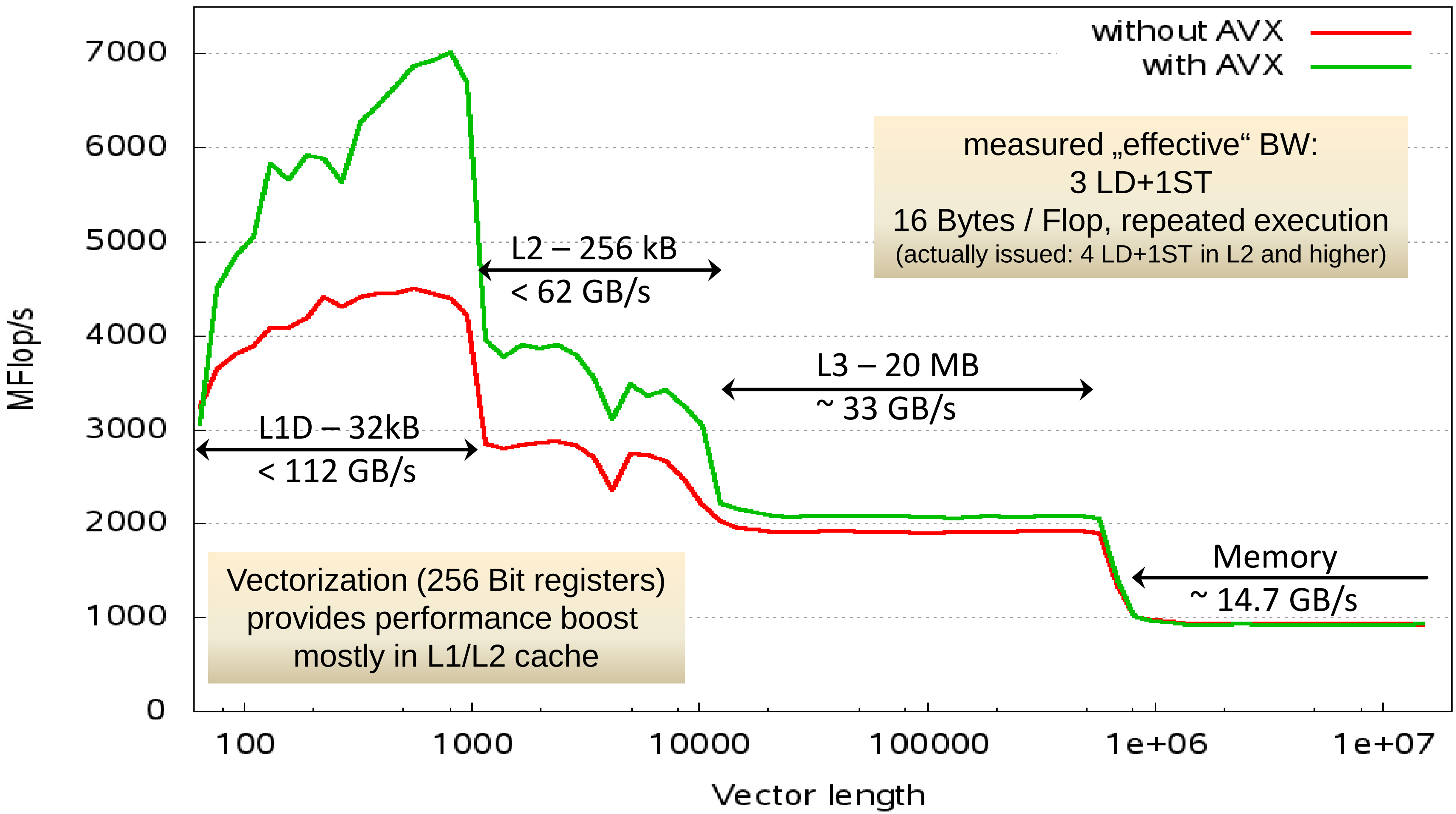
Kernel	Name	Flops	Loads	Stores
$s = s + a_i * b_i$	Scalar Product	2	2	0
$n^2 = n^2 + a_i * a_i$	Norm	2	1	0
$a_i = b_i * s + c_i$	Linked Triad (Stream)	2	2	1
$a_i = b_i * c_i + d_i$	**Vector Triad**	2	3	1

- run repeated iterations for varying vector lengths (working set sizes)

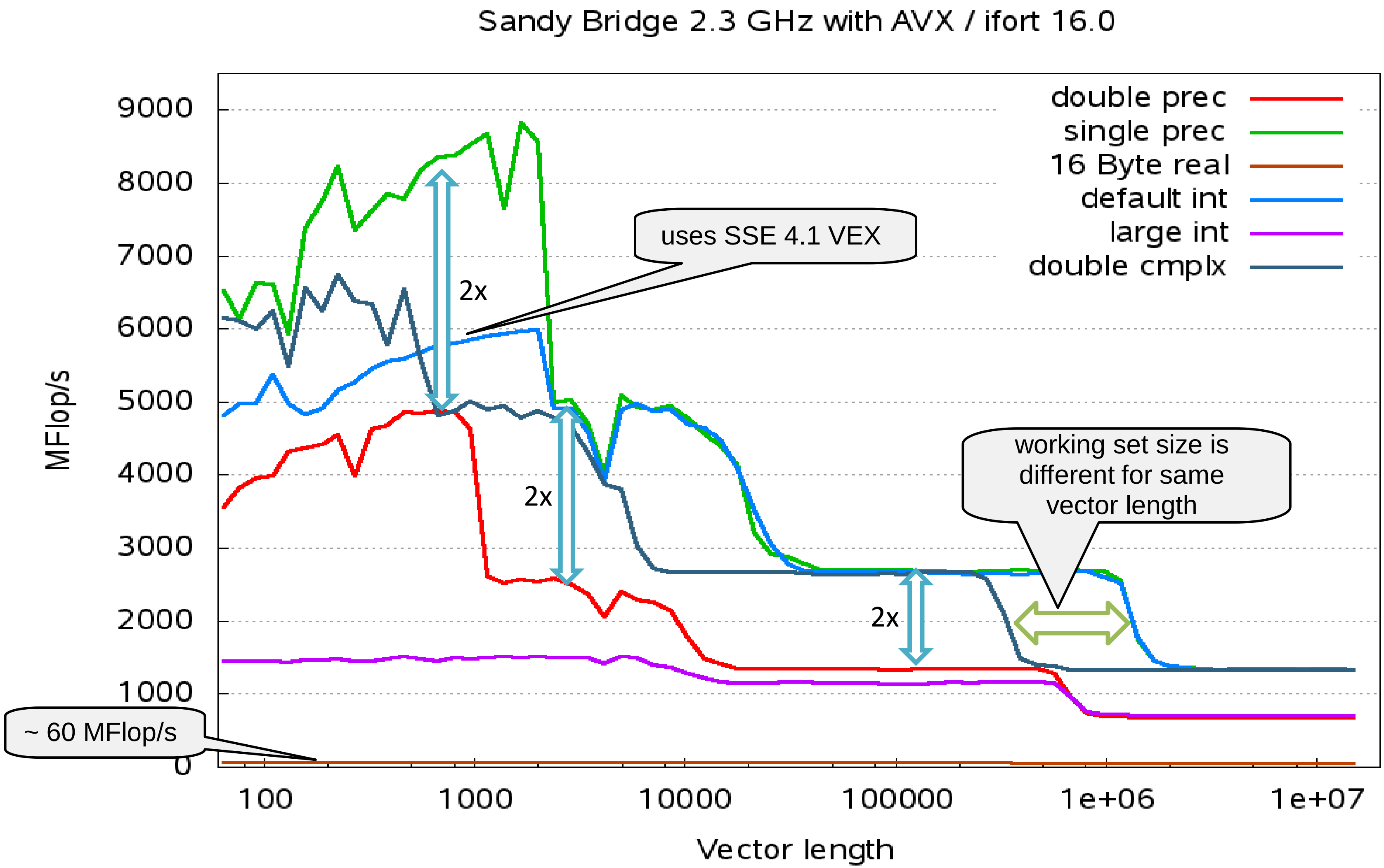
Vector Triad D(:) = A(:) + B(:) * C(:)

- **Synthetic benchmark:** bandwidths of „raw“ architecture, looped version for a **single core**

Sandy Bridge 2.7 GHz / ifort 13.1



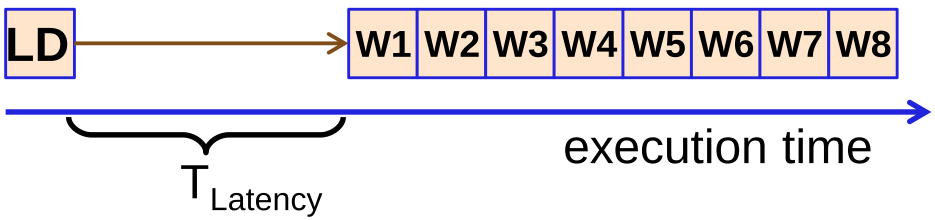
Performance by type and kind



Microprocessor Architecture continued

Loads and Stores

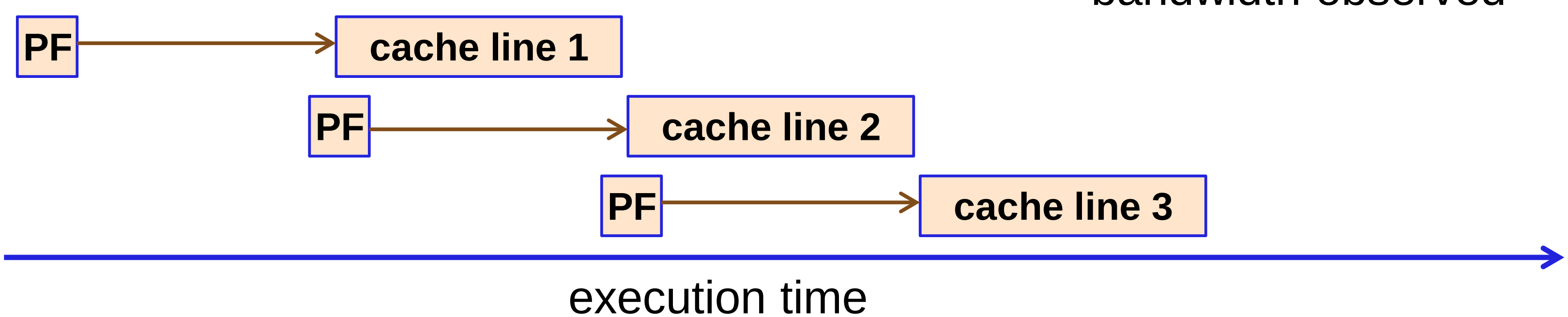
- apply to cache lines



- size: fixed by architecture
  (64, 128 or more Bytes)

Pre-fetch

- avoid latencies when streaming data



- pre-fetches are usually done in hardware
- decision is made according to memory access pattern

Pre-Requisite:

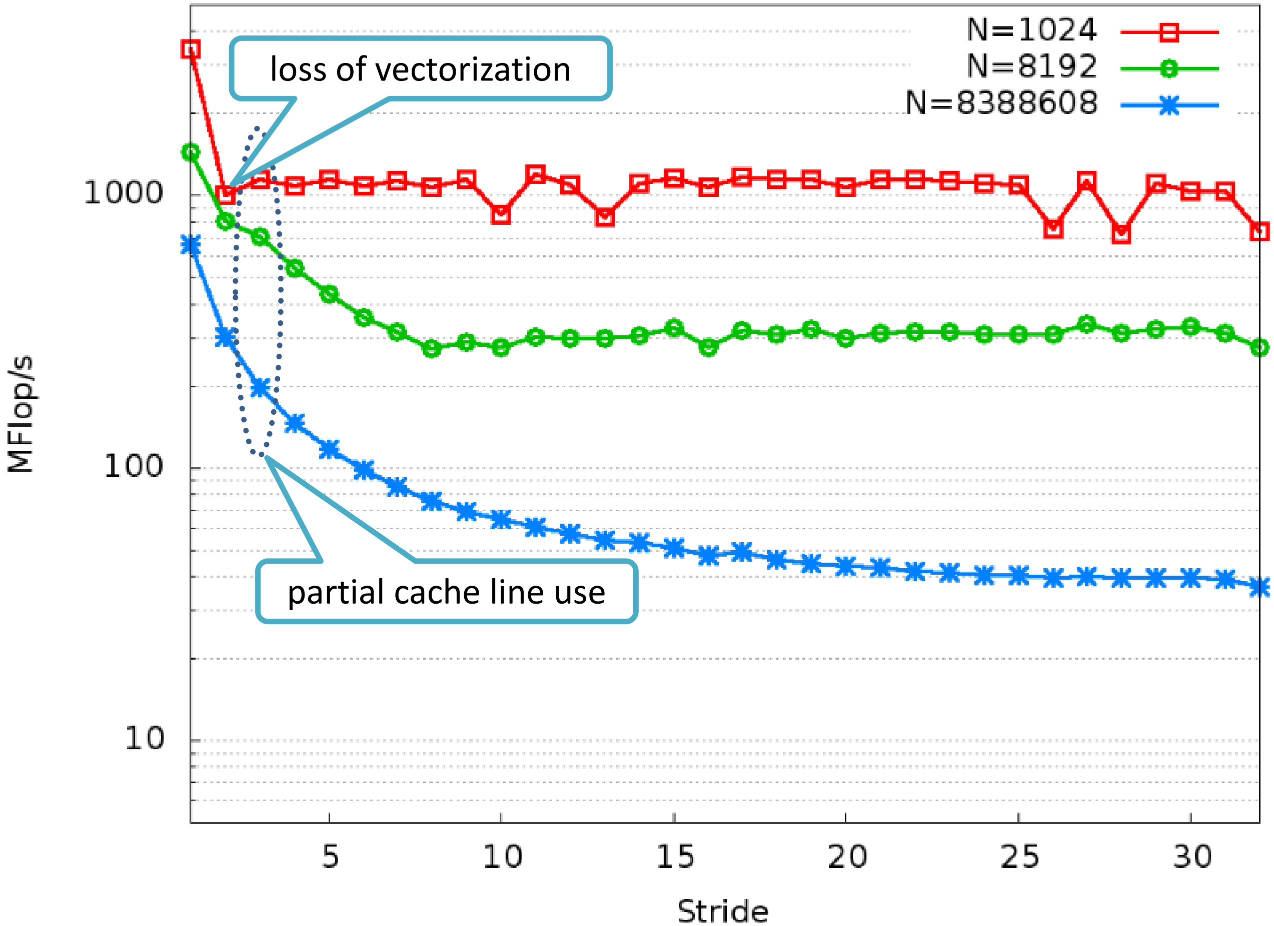
- **spatial** locality
- **violation** of spatial locality:

  if only part of a cache line is used → effective reduction in bandwidth observed

Performance of strided triad on Sandy Bridge
(loss of spatial locality)

```
D(::stride) = A(::stride) + B(::stride)*C(::stride)
```

Example: stride 3



Notes:

- stride known at compile time
- serial compiler optimizations may compensate perfor-mance losses in real-life code

ca. 40 MFlop/s
(remains constant for strides > ~25)

Avoid loss of spatial locality

- **Avoid incorrect loop ordering**

```
REAL :: a(ndim, mdim)

DO i=1, n
  DO j=1, m
     a(i, j) = …
  END DO
END DO
```

jumps through in strides of `ndim`

- **Correct:**

```
REAL :: a(ndim, mdim)

DO j=1, m
  DO i=1, n
     a(i, j) = …
  END DO
END DO
```

innermost loop corresponds to leftmost array index

- **Accessing type components**

```
TYPE(body) :: a(ndim)

DO i=1, n
  … = a(i)%vel(3)
END DO
DO i=1, n
  … = a(i)%pos(3)
END DO
```

effectively stride 8

Array of structures

```
TYPE(body) :: a(ndim)

DO i=1, n
  … = a(i)%mass
  … = a(i)%pos(:)
  … = a(i)%vel(:)
END DO
```

uses 7/8 of cache line

Structures of Arrays

- **Improve vectorizability by**
  - assuring use of contiguous storage sequences of numeric intrinsic type inside objects
- **In general, this requires moving**
  - from arrays of structures to structures of arrays
- **Options in Fortran:**

1. "container" type (with allocatable F03 or pointer F95 components):

```
TYPE :: mbody
  REAL, ALLOCATABLE :: mass(:), &
        pos(:,:), vel(:,:)
END TYPE
```

2. parameterized derived type: F03

```
TYPE :: mbody_pdt(k,l)
  INTEGER, KIND :: k = KIND(1.0)
  INTEGER, LEN :: l
  REAL(KIND=k) :: mass(l), &
        pos(3,l), vel(3,l)
END TYPE
```

compile-time

usually run-time

component size in first dimension is fixed

Memory Layout

Establishing an object

```
TYPE(mbody) :: asteroids
na = ...   ! number of asteroids
ALLOCATE(asteroids%mass(na), &
         asteroids%pos(3,na), ...)
: ! process asteroids
```

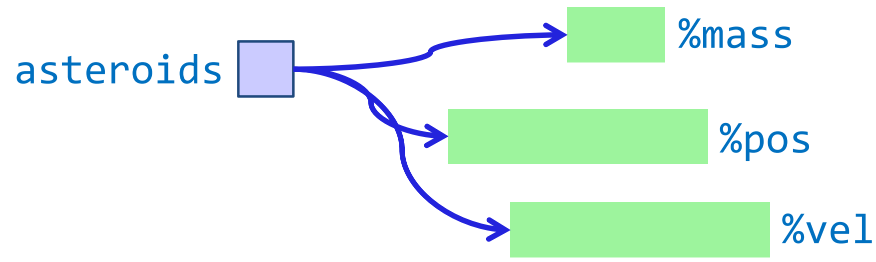
- for mbody, always on the heap

deferred type parameter

```
TYPE(mbody_pdt(l=:)), &
    ALLOCATABLE :: asteroids_pdt
na = ...   ! number of asteroids
ALLOCATE(mbody_pdt(l=na) :: &
    asteroids_pdt)
: ! process asteroids_pdt
```

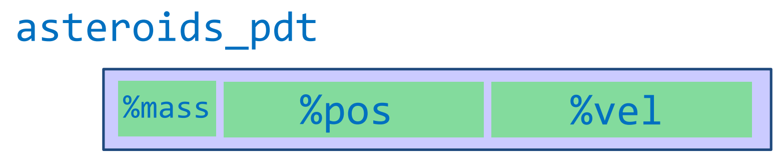
- for mbody_pdt, complete object could also reside on the stack

Scattered object



- vectorization for each component individually

Compact object



- both vectorization and memory streaming for arrays of PDT can be efficiently performed (in theory)

The CONTIGUOUS attribute

- **Avoid non-contiguous access for assumed-shape arrays:**

```
MODULE mod_solver
  IMPLICIT NONE
CONTAINS
  SUBROUTINE process_array_contig(ad)
    REAL, INTENT(inout), CONTIGUOUS :: ad (:,:)
    :
  END SUBROUTINE
END MODULE
```

assures contiguity of dummy argument

- **Expected effect at invocation:**
  - with a contiguous actual argument → passed as usual
    (actual argument: a whole array, a contiguous section of a whole array, or an object with the CONTIGUOUS attribute, …)
  - with a non-contiguous actual argument → copy-in / copy-out
    (performance tradeoff for creating the compactified temporary array depends on problem size and number of calls)

CONTIGUOUS pointers

- **Difference to assumed-shape array**
  - **programmer** is responsible for guaranteeing the contiguity of the target in a pointer assignment
- **Examples:**

```
REAL, POINTER, CONTIGUOUS :: matrix(:,:)
REAL, ALLOCATABLE :: storage(:)
:
ALLOCATE(storage(n*n))
matrix(lb:ub,lb:ub) => storage
diagonal => storage(::n+1)
```

  - first pointer assignment is legitimate because whole allocated array `storage` is contiguous
  - if contiguity of target is not known, need to check via intrinsic:

```
IF ( is_contiguous(other_storage) ) THEN
  matrix(lb:ub,lb:ub) => other_storage
ELSE
  …
```

(with possibly new values for `lb, ub`)

- **Language design was from the beginning such that processor's optimizer not inhibited**
  - loop iteration variable is not permitted to be modified inside loop body
    → enables register optimization (provided a local variable is used)
  - aliasing rules (no overlap between dummy argument and some other accessible variable if at least one is modified)
    → enables optimization of array operations (based on dependency analysis)
- **With modern Fortran**
  - extension of the existing aliasing rules for POINTER and ALLOCATABLE objects F95 , and for coarrays F18
- **Other languages have caught up**
  - e.g. beginning with C99, C has the `restrict` keyword for pointers
    → similar aliasing rules as for Fortran

ELEMENTAL procedures

- **Declaration:**
  - ELEMENTAL prefix

```
MODULE elem_stuff
CONTAINS
  ELEMENTAL subroutine swap(x, y)
    REAL, INTENT(inout) :: x, y
    REAL :: wk
    wk = x; x = y; y = wk
  END SUBROUTINE swap
END MODULE
```

  - all dummy arguments (and function result if a function) must be scalars
  - an interface block is required for an external procedure
  - elemental procedures are also PURE

F08 introduces an IMPURE attribute for cases where PURE is inappropriate

Invoking an ELEMENTAL procedure

- **Actual arguments (and possibly function result)**
  - can be all scalars or all conformable arrays

```
USE elem_stuff
REAL :: x(10), y(10), z, zz(2)
:   ! define all variables
CALL swap(x, y)         ! OK
CALL swap(zz, x(2:3)) ! OK
CALL swap(z, zz)        ! invalid
```

  - execution of subroutine applies for every array element
- **Further notes:**
  - vectorization potential (maybe using OpenMP SIMD construct)
  - many intrinsics are elemental
  - some array constructs: subprogram calls in body may need to be elemental

WHERE statement and construct
(„masked operations“)

- **Execute array operations only for a subset of elements**
  - defined by a logical array expression e.g.,

```
WHERE ( a > 0.0 ) a = 1.0/a
```

  - general form:

```
WHERE ( x ) y = expr
```

    wherein **x** must be a logical array expression with the same shape as **y**.
  - **x** is evaluated first, and the evaluation of the assignment is only performed for all index values for which **x** is true.

- **Multiple assignment statements**
  - can be processed with a **construct**

```
WHERE ( x )
  y1 = …
  y2 = …
  y3 = …
[ ELSEWHERE [( z )]
  y4 = … ]
END WHERE
```

optional only for **final** elsewhere block

  - same mask applies for every assignment
  - **y4** is assigned for all elements with **.not. x .and. z**

Assignment and expression in a WHERE statement or construct

- **Assignment may be**
  - a defined assignment (introduced later) if it is elemental
- **Right hand side**
  - may contain an elemental function reference. Then, masking extends to that reference
  - may contain a non-elemental function reference. Masking does **not** extend to the argument of that reference

```
WHERE (a > 0.0) &
          a = SQRT(a)
```

`sqrt()` is an elemental intrinsic

```
WHERE (a > 0.0) &
          a = a / SUM(LOG(a))
```

`sum()` is an non-elemental intrinsic
→ all elements must be evaluated in `log()`

  - array-valued non-elemental references are also fully evaluated **before** masking is applied

FORALL statement

- **Parallel semantics**
  - of array element assignment

```
FORALL (i=1:n, j=5:m:2) a(i, j) = b(i) + c(j)
```

    expression can be evaluated in any order, and assigned in any order of the index values

  - conditional array element assignment

```
FORALL (i=1:n, c(i) /= 0.0) b(i) = b(i)/c(i)
```

  - more powerful than array syntax – a larger class of expressions is implicitly permitted

```
FORALL (i=1:n) a(i,i) = b(i)*c(i)
```

FORALL construct

- **Multiple statements to be executed**
  - can be enclosed inside a construct

```
FORALL (i=1:n, j=1:m-1)
  a(i,j) = real(i+j)
  where (d(i,:,j) > 0) a(i,j) = a(i,j) + d(i,:,j)
  b(i,j) = a(i,j+1)
END FORALL
```

effectively, an array assignment

Flow dependency for array a

- **Semantics:** each statement is executed for **all** index values **before** the next statement is initiated

  in the example, the third statement is conforming if a(:,m) was defined prior to the FORALL construct; the other values of a are determined by the first statement.

- this limits parallelism to each individual statement inside the block

Further notes on FORALL

- **Permitted statement types inside a FORALL statement or construct**
  - array assignments
    (may be defined assignment)
  - calls to PURE procedures
  - `WHERE` statement or construct
  - `FORALL` statement or construct
  - pointer assignments
    (discussed later)

- **Issues with FORALL:**
  - implementations often (need to) generate many array temporaries
  - statements are usually not parallelized anyway
  - performance often worse than that of normal DO loop

→ **Recommendation:**
  - **do not use `FORALL`** in performance critical code sections

F18 **flags FORALL obsolescent**

The DO CONCURRENT construct

F08

Improved parallel semantics

- requirement on program: statements must not contain **dependencies** that inhibit parallelization
- syntax: an extension of the standard DO construct

```
DO CONCURRENT ( i=1:n, j=1:m, i<=j )
  a(i, j) = a(i, j) + alpha * b(i, j)
END DO
```

optional logical mask that curtails the iteration space

- constraints prevent introducing dependencies: checked by compiler. **Impermissible:** `CYCLE` or `EXIT` statements that exit the construct, impure procedure calls

Permission / Request to compiler for

- parallelizing loop iterations, and/or
- vectorizing / pipelining loop iterations

Example: Intel Fortran will perform multi-threading if the `-parallel` option is specified

Examples

Incorrect usage

```
DO CONCURRENT (i=1:n, j=1:m)
  x = a(i, j) + …
  b(i, j) = x * c(j, i)
  if (j > 1) a(i, j) = b(i, j-1)
END DO
```

- flow dependencies for real scalar **x** and **b** make correct parallelization impossible
- note that **x** is updated by iterations different from those doing references

Correct usage

```
DO CONCURRENT (i=1:n, j=1:m)
  BLOCK
    REAL :: x
    x = a(i, j) + …
    b(i, j) = x * c(j, i)
  END BLOCK
END DO
DO CONCURRENT (j=2:m)
  a(:, j) = b(:, j-1)
END DO
```

per-iteration variable is created

- performance is implementation-dependent

F18 extensions

- **Clauses for locality specification**

```fortran
REAL :: x
:
DO CONCURRENT (i=1:n, j=1:m) &
     LOCAL(x) SHARED(a, b, c)
   x = a(i, j) + …
   b(i, j) = x * c(j, i)
END DO
DO CONCURRENT (j=2:m)
  a(:, j) = b(:, j-1)
END DO
```

- guarantees that per-iteration variable **x** is created

- **Table of locality specifications**

clause	semantics
`LOCAL`	create per-iteration copy of variable inside construct
`LOCAL_INIT`	same as local, but value from variable prior to execution is copied in
`SHARED`	references and definitions are to original variable
`DEFAULT(NONE)`	force declaration of locality spec for all entities in construct

Some I/O extensions

Derived type I/O

- **Statements like**

```
TYPE(...) :: obj
:
WRITE(iu) obj
WRITE(iu, FMT=...) obj
```

  - will work if `obj` is statically typed and has static type components
- **They will not work in following situations:**
  - the type has POINTER or ALLOCATABLE type components, or
  - the object is polymorphic.

  In both cases, the I/O transfer statements are rejected at compile time
- **Therapy:**
  - overload I/O statements with user-defined routines F03

I/O procedure interfaces

- **Two signatures exist:**

these are formal interfaces only

```
SUBROUTINE formatted_dtio   (dtv,unit,iotype,v_list,iostat,iomsg)
SUBROUTINE unformatted_dtio (dtv,unit,                iostat,iomsg)
```

- `dtv`
  - scalar of derived type
  - polymorphic iff type is extensible
  - `INTENT` depends on semantics

i.e. READ or WRITE

- `unit`
  - `INTEGER, INTENT(in)` – describes I/O unit or is negative for internal I/O
- `iotype` (formatted only)
  - `CHARACTER, INTENT(in)` - with values `'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(*), INTENT(inout)` - explanation for failure if `iostat` nonzero

Bind procedure to type

- **Assume you have implemented following procedures:**
  - `write_fmt_mbody(...)` for formatted writing
  - `read_unfmt_mbody(...)` for unformatted reading

  with the interfaces given on the previous slide
- **Generic type-bound procedures:**

```
TYPE :: mbody
  : ! allocatable components
CONTAINS
  PROCEDURE :: write_fmt_mbody, read_unfmt_mbody
  GENERIC :: READ(UNFORMATTED) => read_unfmt_mbody
  GENERIC :: WRITE(FORMATTED) => write_fmt_mbody
END TYPE
```

- **Notes:**
  - inside a formatted I/O procedure, only non-advancing transfers are done
  - no record termination is done, and the `REC=` and `POS=` specifiers are not permitted
  - you can override the TBPs for an extension of `mbody`

Invoke through I/O statements

- **Implicit invocation**

```
TYPE(mbody) :: asteroids
: ! connect files to units ir, iw
READ(ir) asteroids

WRITE(iw, FMT='(DT ”Mbody” (12,5))', IOSTAT=stat) asteroids
```

dispatches to `read_unfmt_mbody()`

dispatches to `write_fmt_mbody()`

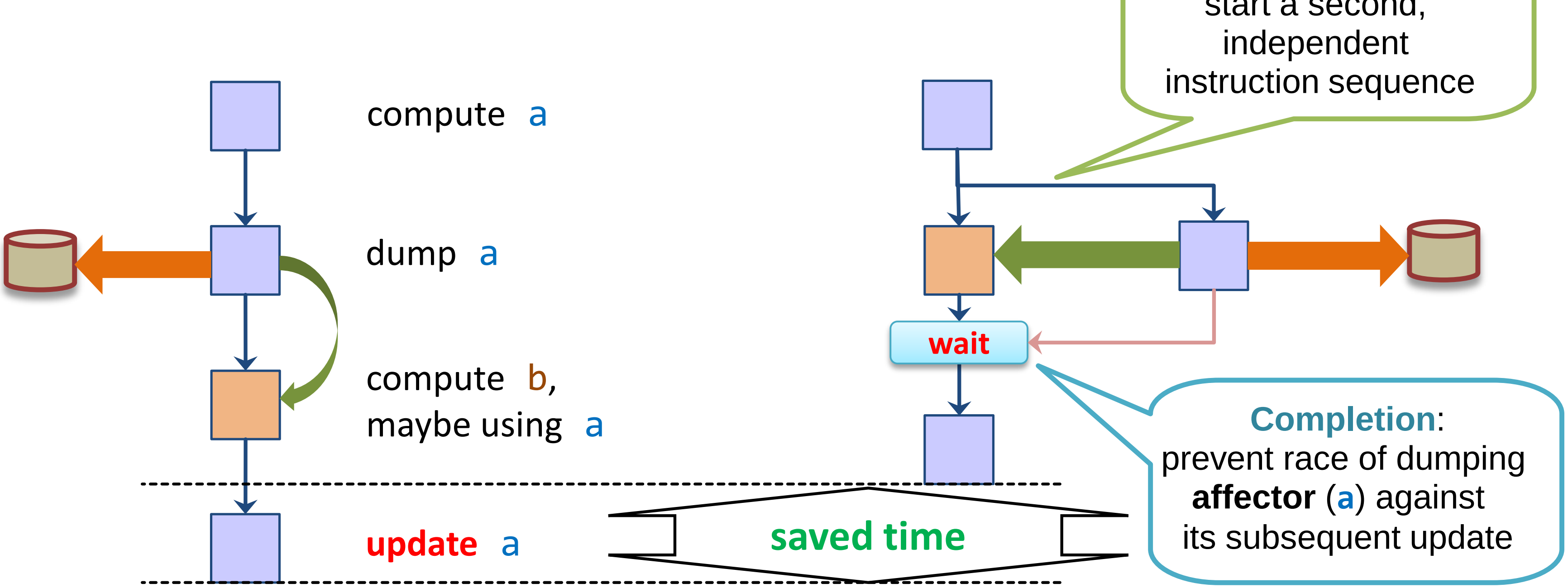
Available in **`iotype`**
Empty string if omitted

Available in **`v_list`**
Empty array if omitted

- **Both `iotype` and `v_list` are available to the programmer of the I/O subroutine**
  - they determine further parameters of I/O as programmer sees fit

Asynchronous processing

- **An idea for performance tuning:**
  - overlap computation with independent data transfers



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

The ASYNCHRONOUS attribute:
Contractual obligations between initiation and completion

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

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

  - here: for an assumed-shape array dummy argument
  - sometimes also implicit

- **Processor:**
  - do not perform code motion of references and definitions of affector across initiation or completion procedure
  - if one of them is not identifiable, code motion across procedure calls is generally prohibited, even if the affector is not involved in any of them
- **Constraints for dummy arguments**
  - assure that no copy-in/out can happen to affectors

F03

Scenario 1: asynchronous I/O

Example: non-blocking READ

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

no prefetches on **a** here

Actual asynchronous execution

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

Ordering requirements

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

`ID` specifier

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

INQUIRE

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

Scenario 2: non-blocking MPI calls

- **Non-blocking receive - equivalent to a READ operation**

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

asynchronous execution is limited to BLOCK

permitted, but may perform better outside the BLOCK

- **Likely a good idea to avoid call stacks with affector arguments**
  - violations of contract or missing attribute can cause quite subtle bugs that surface rarely

Unit testing
Code coverage
Documentation

Definition of the term

- **"A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work."**
  -- The Art of Unit Testing

- **In the Fortran context, a unit of work is a Fortran procedure**

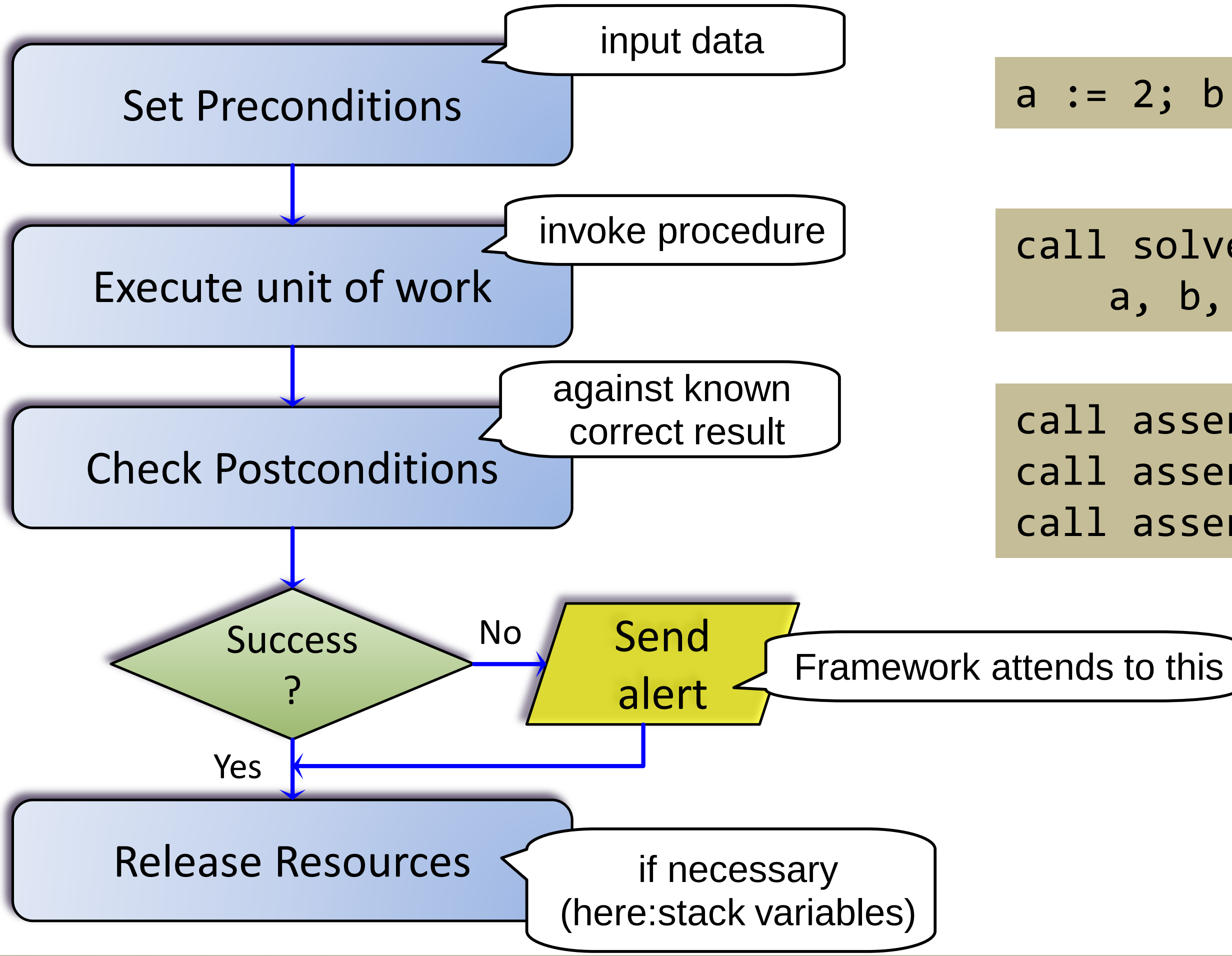
- **Unit testing framework discussed here:**
  - pFUnit by Tom Clune (NASA)
    http://sourceforge.net/projects/pfunit/
  - a full-featured framework (written mostly in Fortran)
  - slides presented here are strongly influenced by his tutorial material

Desired properties of tests

- **Narrow/specific**
  - failure of a test localizes defect to small section of code
- **Orthogonal to other tests**
  - each defect causes failure in one or only a few tests
- **Complete**
  - all functionality is covered by at least one test
  - any defect is detectable
- **Independent - No side effects**
  - no STDOUT; temp files deleted; etc
  - order of tests has no consequence
  - failing test does not terminate execution
- **Frugal**
  - execute quickly
  - minimal resource usage

Unit testing procedure

- **General scheme**



- **Example:** $ax^2 + bx + c = 0$
  - real quadratic solver

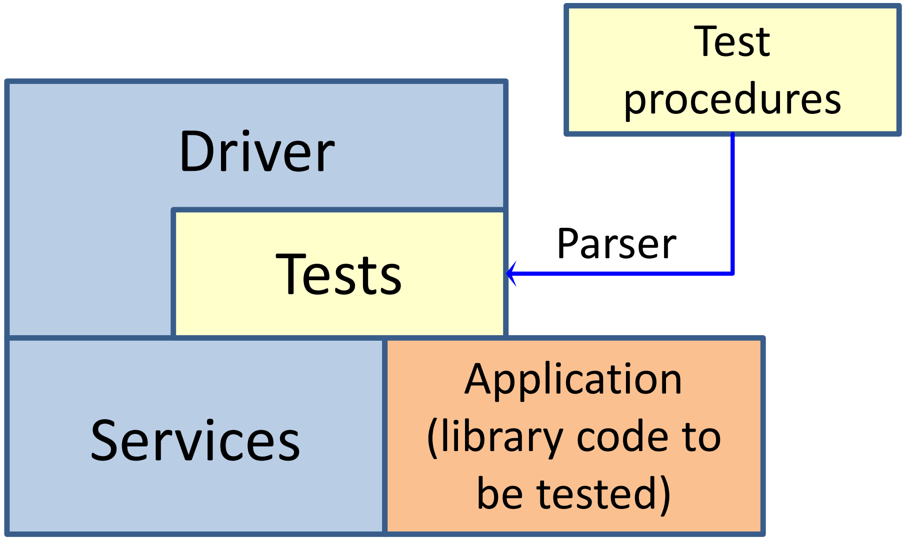
```
a := 2; b := 2; c := -1.5
```

```
call solve_quadratic( &
        a, b, c, n, x1, x2 )
```

```
call assertEqual (2, n)
call assertEqual(-1.5, x1)
call assertEqual(0.5, x2)
```

Testing framework

- **Architecture:**



- **`assertEqual` is part of the framework**
- **Test procedures are usually written using macros**
  - not standard-conforming Fortran
  - must be preprocessed:

```
@assertEqual (2, n)
```

will be expanded to

```
call assertEqual (2, n , location = SourceLocation ('testQuadratic.pf', 5) )
if ( anyExceptions() ) return
# line 6 “testQuadratic.pf"
```

source file name and line information is added

Complete example test procedure

- File `testQuadratic.pf`

```
@test
subroutine testQuadratic ()
  use pFUnit_mod
  use mod_quadratic
  real :: a, b, c, x1, x2
  integer :: n
  a = 2; b = 2; c = -1.5
  call solve_quadratic( &
        a, b, c, n, x1, x2 )
  @assertEqual ( 2, n )
  @assertEqual ( -1.5, x1 )
  @assertEqual ( 0.5, x2 )
end subroutine testQuadratic
```

obligatory

  - binds together application and framework code

- pFUnit driver
  - requires additional include file that registers tests with the driver
  - file **`testSuites.inc`**:

```
! other active lines may exist
ADD_TEST_SUITE(testQuadratic_suite)
```

case sensitive!

  - one suite is generated per file or module
  - **naming convention** for test suites:

1. If the test procedure is a module, **`<module_name>_suite`**
2. Explicitly determined by **`@suite=<name>`** in test procedure
3. Otherwise, **`<file_name>_suite`**

Building a testing executable

Makefile rules

```
.PHONY: tests clean
%.F90 : %.pf
        $(PFUNIT)/bin/pFUnitParser.py $< $@ -I.
TESTS = $(wildcard *.pf)

%.o : %.F90
        $(FC) -c $< -I $(PFUNIT)/mod

SRCS = $(wildcard *.F90)
OBJS = $(SRCS:.F90=.o) $(TESTS:.pf=.o)
DRIVER = $(PFUNIT)/include/driver.F90

tests.x : $(DRIVER) $(OBJS)
        $(FC) -o $@ -I$(PFUNIT)/mod $^ -L$(PFUNIT)/lib -lpfunit -I.
tests : tests.x
        ./tests.x
clean :
        $(RM) *.o *.mod *.x *~
```

PFUNIT must be set to pFUnit installation path

this is the parser run on *.pf files

Some further remarks

- **The pFUnit parser imposes following limitations:**
  - each annotation must be on a single line
  - no end-of-line comment characters
  - comment at beginning of line deactivates an annotation
- **Some restrictions on syntax for intermingled Fortran:**
  - only supports free-format
    (fixed-format application code is OK.)
- **Test procedure declarations must be on one line**

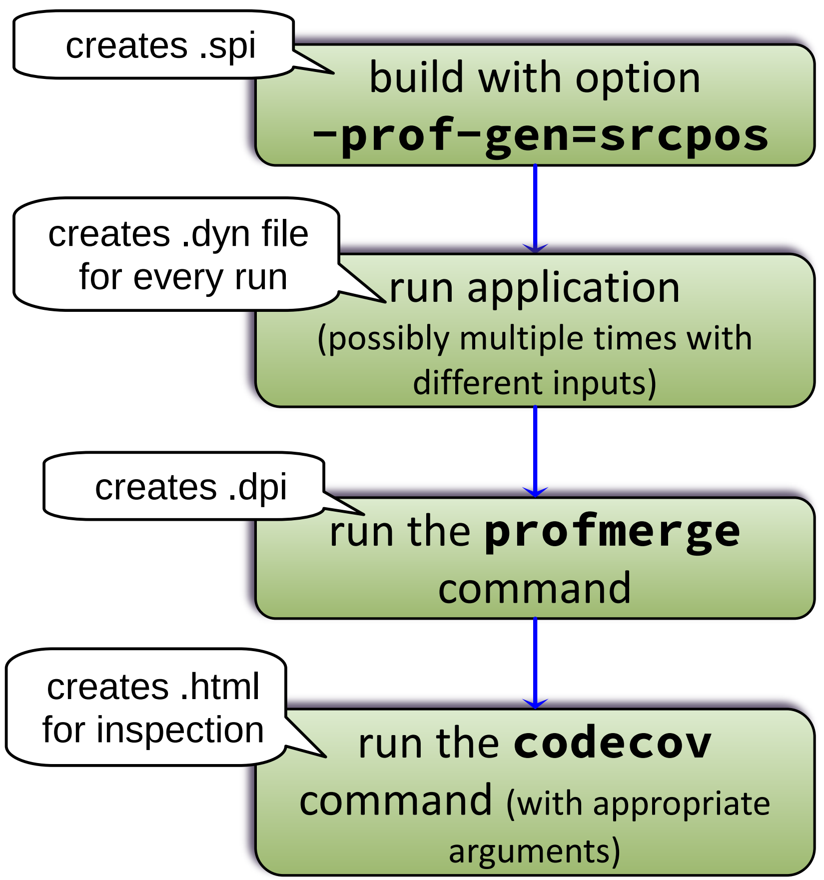
```
@test
subroutine &
  testQuadratic ()
```

not permitted

- **Multiple tests in a single file are possible**
  - each subroutine must be prepended by `@test` macro

Code coverage

- **How can I assure testing is complete?**
  - Intel compilers support code coverage analysis



Example output (1)

.spi file for library code, **not** the framework code

```
codecov -prj testQuadratic -spi ../pgopti.spi -dpi pgopti.dpi
```

Coverage Summary of testQuadratic

Files				Functions				Blocks			
total	cvrd	uncvrd	cvrg%	total	cvrd	uncvrd	cvrg%	total	cvrd	uncvrd	cvrg%
2	1	1	**50.00**	2	1	1	**50.00**	39	8	31	**20.51**

Covered Files in testQuadratic

Name	Functions			Blocks		
	total	cvrd	cvrg%	total	cvrd	cvrg%
mod_quadratic.f90	1	1	100.00	17	8	47.06

Example output (2)

```
  real :: disc, r, tmp

  disc = b**2 - 4 * a * c

  if (a == 0.0) then
     if (b == 0.0) then
        n = 0   ! c == 0.0 actually would imply an infinity of solutions
        return
     else
        n = 1
        x1 = - c / b
     end if
     return
  end if

  if (disc > 0.0) then
     n = 2
     if (b == 0.0) then
        r = abs(sqrt(disc) / (2.0 * a))
        x1 = -r
        x2 = r
     else
        tmp = -0.5 * (b + sign(sqrt(disc),b))
        x1 = tmp / a
        x2 = c / tmp
        if (x1 > x2) then
           tmp = x1
           x1 = x2
           x2 = tmp
        end if
     end if
  elseif (disc == 0.0) then
     n = 1
     x1 = -0.5 * b / a
  else
     n = 0
  end if
end subroutine solve_quadratic
end module
```

code blocks that were **not executed** are marked yellow

Generation of documentation (1)

- **Basic idea:**
  - documentation should be directly generated from source code
  - annotations by programmer
  - reduce maintenance effort
  - increase chance of documentation being consistent with implementation
- **One possible tool: Doxygen**
  - has support for many languages, including Fortran

- **Step 1: Generate template**

```
doxygen -g my_doxy.conf
```

- **Step 2: Edit the file `my_doxy.conf`**
  - following settings are of interest:

PROJECT_NAME	(your choice)
OPTIMIZE_FOR_FORTRAN	(set to YES)
EXTRACT_ALL	
EXTRACT_PRIVATE	
EXTRACT_STATIC	
INPUT	(other source directories)
FILE_PATTERNS	
HAVE_DOT	(set to YES)
CALL_GRAPH	(set to YES)
CALLER_GRAPH	(set to YES if you want)

- **Step 3: Run tool**

```
doxygen my_doxy.conf
```

  - subdirectories `html` and `latex` are created (documentation formats)

Generation of documentation (2)

- **Annotation of source code**
  - special tags indicate what kind of entities are described
  - bulleted lists
  - LaTeX style formulas (requires a LaTeX installation)
  - many special commands to change default generation mechanisms (or work around bugs)
- **Output formats**
  - HTML and LaTeX (→ PDF) by default
  - others are possible
- **Example**
  - see the `examples/doxygen` folder for demonstration code
- **Web page / Documentation**

http://www.stack.nl/~dimitri/doxygen/

- **Alternative**
  - FORD

https://github.com/cmacmackin/ford

  - more specifically designed for Fortran
  - similar in spirit, though

Coarrays

Partitioned Global Address Space

Fortran and Parallelism

Design target for PGAS extensions:

smallest changes required to convert Fortran
into a robust and efficient parallel language

- add only a few new rules to the language
- provide mechanisms to allow

explicitly parallel execution: **SPMD style** programming model

data distribution: **partitioned memory** model

synchronization against race conditions

memory management for dynamic shared entities

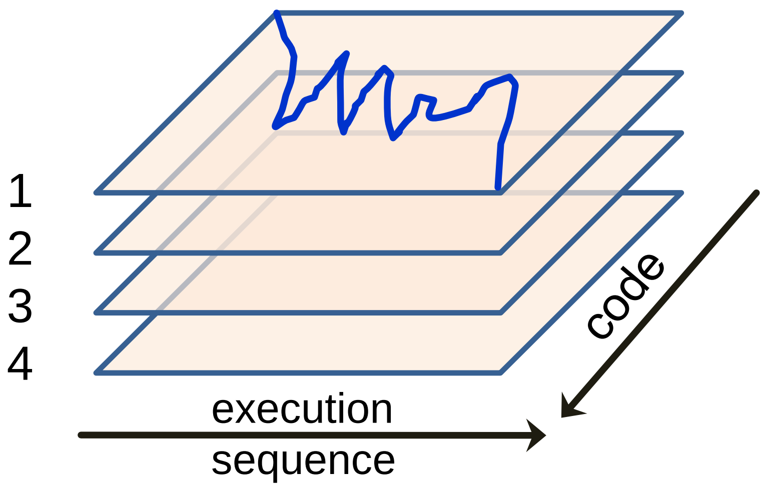
Standardization efforts:

- F08 basic coarray features
- F18 significant extension of parallel semantics

gfortran implements a small subset

Execution model: concept of image

- **Going from serial to parallel execution**
  - CAF - **images**:



  - image counts between 1 and number of images

- **Replicate single program a fixed number of times**
  - set number of replicates at **compile** time or at **execution** time
  - asynchronous execution – **loose** coupling unless program-control-led synchronization occurs
- **Separate set of entities on each replicate**
  - program-controlled exchange of data
  - necessitates synchronization

Optional slide

Comparison with other parallelization methods

lrz

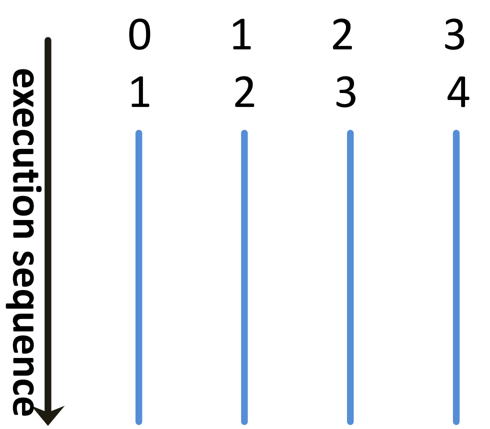
ratings: 1-low 2-moderate 3-good 4-excellent

	MPI	OpenMP	Coarrays	UPC
Portability	yes	yes	yes	yes
Interoperability (C/C++)	yes	yes	no	yes
Scalability	4	2	1-4	1-4
Performance	4	2	2-4	2-4
Ease of Use	1	4	2.5	3
Data parallelism	no	partial	partial	partial
Distributed memory	yes	no	yes	yes
Data model	fragmented	global	fragmented	global
Type system integrated	no	yes	yes	yes
Hybrid parallelism	yes	partial	(no)	(no)

Coarray Fortran (and PGAS in general):
good scalability for fine-grain parallelism in distributed memory systems will require use of special interconnect hardware features

Interoperation with MPI

- **Nothing is formally standardized**
- **Existing practice:**
  - each MPI task is identical with a coarray image



result of calling `MPI_Comm_rank()`

`this_image()`

no guarantee on ordering, though

obtained from call to `MPI_Initialized()`

```
PROGRAM with_mpi
  USE mpi_f08
  : ! further declarations, including coarrays
  IF (.not. initialized) CALL MPI_Init()
  : ! code with both MPI calls and
  : ! coarray communication / synchronization
  CALL MPI_Finalize()
END PROGRAM
```

implementation may either want this or not like this

Simplest possible program

- **Intrinsic functions for image management**

```
PROGRAM hello
  IMPLICIT NONE
  WRITE(*, '(''Hello from image '',i0, '' of '',i0)') &
        this_image(), num_images()
END PROGRAM
```

between **1** and **num_images()**

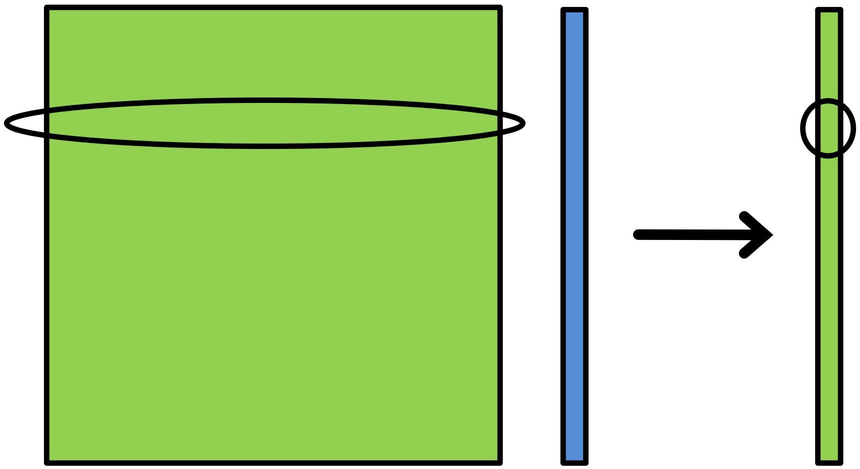
non-repeatably unsorted output
if multiple images are used

- **num_images()**
  - returns number of images (set by environment) - default integer
- **this_image()**
  - generic intrinsic. The form without arguments returns a number between 1 and **num_images()** - default integer
- **Implications**
  - define data distribution / implement trivially parallel algorithms

A more elaborate example: Matrix-Vector Multiplication

$$\sum_{j=1}^{n} M_{ij} \cdot v_j = b_i$$

- **Basic building block for many algorithms**



  - independent collection of scalar products

- **Serial calculation**
  - typically uses an optimized BLAS routine (SGEMV)

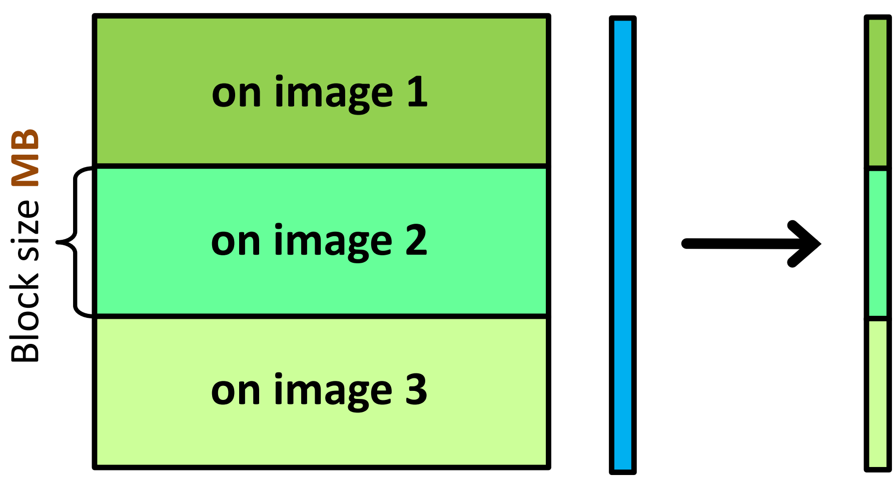
```
INTEGER, PARAMETER :: N = …
REAL :: Mat(N, N), V(N)
REAL :: B(N)   ! result

DO icol=1,N
  DO irow=1,N
    Mat(irow,icol) = &
           matval(irow,icol)
  END DO
  V(icol) = vecval(icol)
END DO
CALL sgemv('n',N,N,1.0,
           Mat,N,V,1,0.0,B,1)
```

initialize matrix and vector

Data decomposition

Block row distribution:

- calculate only a block of B on each image (but that completely)
- the shading indicates the assignment of data to images
- blue: data are replicated on all images



Alternatives exist:

- cyclic, block-cyclic
- column, row and column

Memory requirement:

- (n^2 + n) / <no. of images> + n words per image/thread
- load balanced (same computational load on each task)

Modified declarations:

```
REAL :: Mat(MB, N), V(N)
REAL :: B(MB)
```

Assumption: MB == N / (no. of images)

- dynamic allocation more flexible
- if mod(N, no. of images) > 0, conditioning is required

Work sharing the initialization and the M*v processing

"Fragmented data" model

- need to calculate **global** row index from local iteration variable (or vice versa)

```
DO icol=1,N
 DO i=1,MB
   irow = (this_image() - 1) * MB + i
   Mat(i,icol) = matval(irow,icol)
 END DO
 V(icol) = vecval(icol)
END DO

CALL sgemv('n',MB,N,1.0,Mat,MB,V,1,0.0,B,1)
```

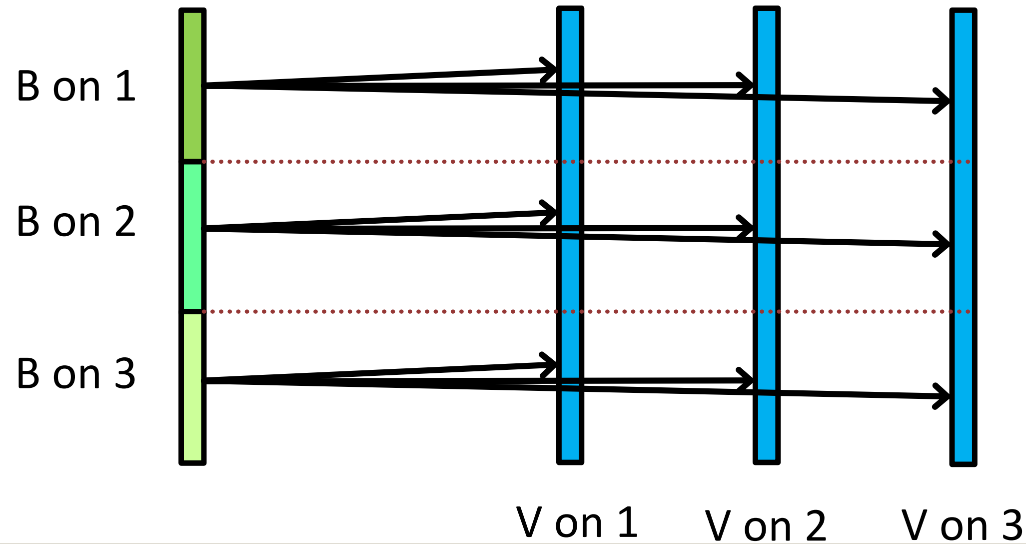
`i` is image-local index;
need to calculate global index `irow`

each image:
works on its own, private instances of Mat, V, B

- degenerates into serial version of code for 1 image
- generalization needed for other decomposition scenarios

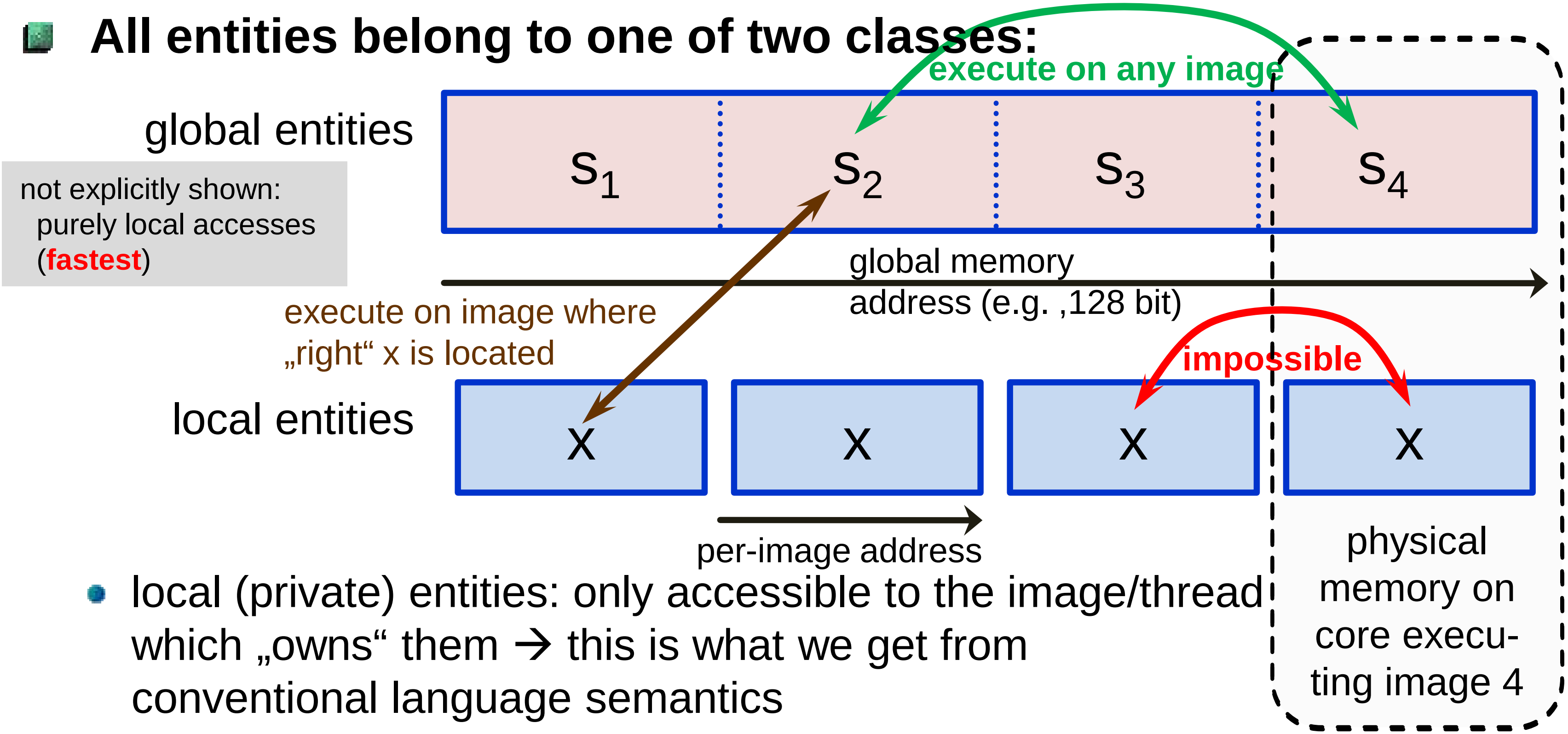
Illustrating the need for communication

- **Open issue:**
  - iterative solvers require **repeated** evaluation of matrix-vector product
  - but the result we received is distributed across the images
- **Therefore, a method is needed**
  - to **transfer** each B to the appropriate portion of V on all images



PGAS data and memory model

- **All entities belong to one of two classes:**



- local (private) entities: only accessible to the image/thread which „owns“ them → this is what we get from conventional language semantics
- global (**shared**) entities in partitioned global memory: objects declared on and physically assigned to one image/thread may be accessed by any other one
- allows implementation for distributed memory systems

the term „shared“:
→ slightly **different** semantics than in OpenMP

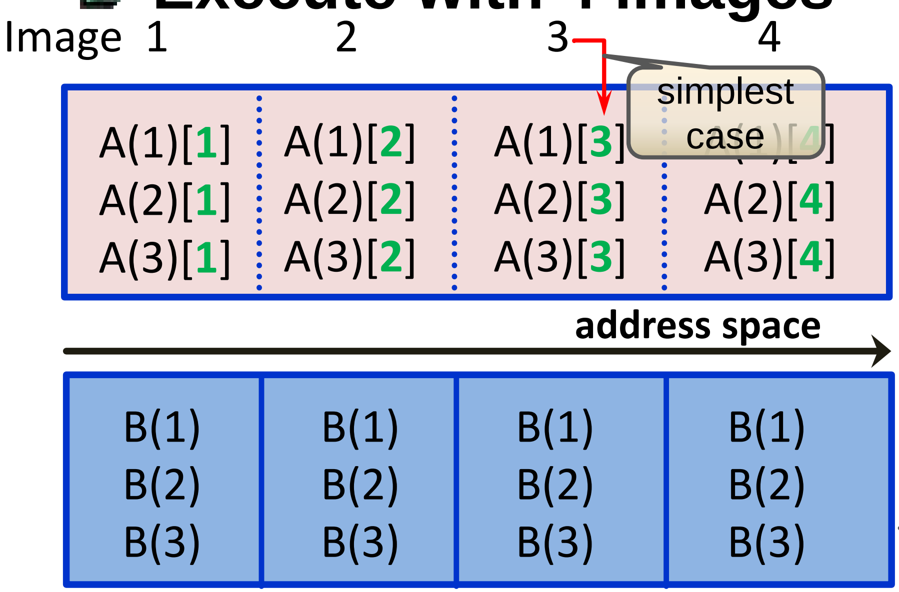
Declaration of coarrays / shared entities
(simplest case)

Coarray declaration

- **symmetric** objects

```
INTEGER :: b(3)
INTEGER :: a(3)[*]
```

Execute with 4 images



- one-to-one mapping of **coindex** to image index

Explicit attribute

- equivalent declaration:

```
INTEGER, CODIMENSION[*] :: a(3)
```

- a scalar coarray:

```
INTEGER, CODIMENSION[*] :: s
```

Difference between A and B?

Inter-image communication: coindexed access

Pull (Get)

```
IF (this_image() == p) &
        b = a(:)[q]
```

sectioning is obligatory

a **coindexed** reference

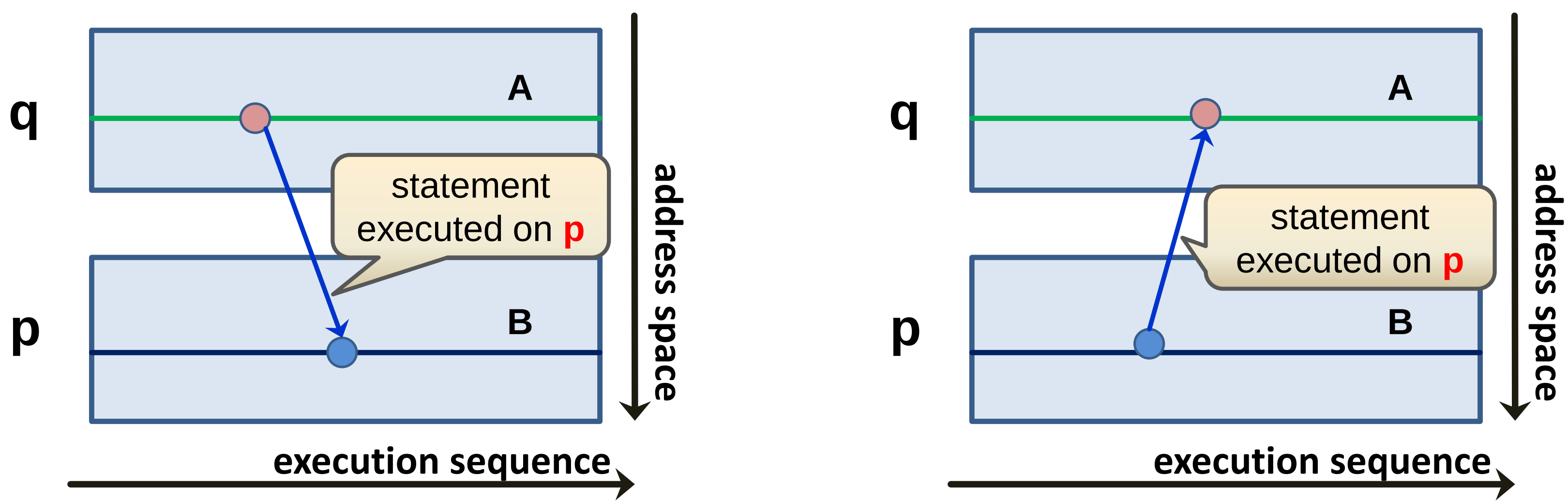
Push (Put)

```
IF (this_image() == p) &
        a(:)[q] = b
```

a **coindexed** definition

assumption: **p** and **q** have the same value on all images, respectively

- **one-sided communication** between images **p** and **q**



Local accesses to coarrays

- **Design aim for non-coindexed accesses:**
  - should be optimizable as if they were local entities

```
INTEGER :: a(3)[*]
INTEGER :: i
a(:) = (/ … /)
:
i = a(3) + …
:
CALL my_proc(a, …)
```

```
a(:)[this_image()] = (/ … /)
```

same meaning, but likely slower execution speed

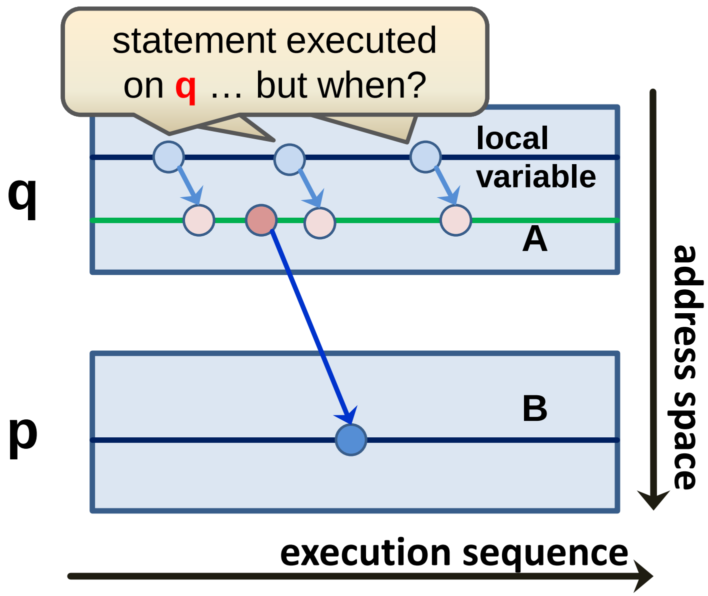
permitted: interface of **`my_proc`** declares dummy argument corresponding to **`a`** as **`REAL :: x(:)`** (not a coarray)

- **Explicit coindexing:**
  - indicates to programmer that communication is happening
  - **distinguish:** coarray (a) ↔ coindexed entity (a[p])
  - cosubscripts must be **scalars** of type integer

Synchronization requirements

- **Asynchronous execution**

```
a = …
IF (this_image() == p) &
        b = a(:)[q]
```

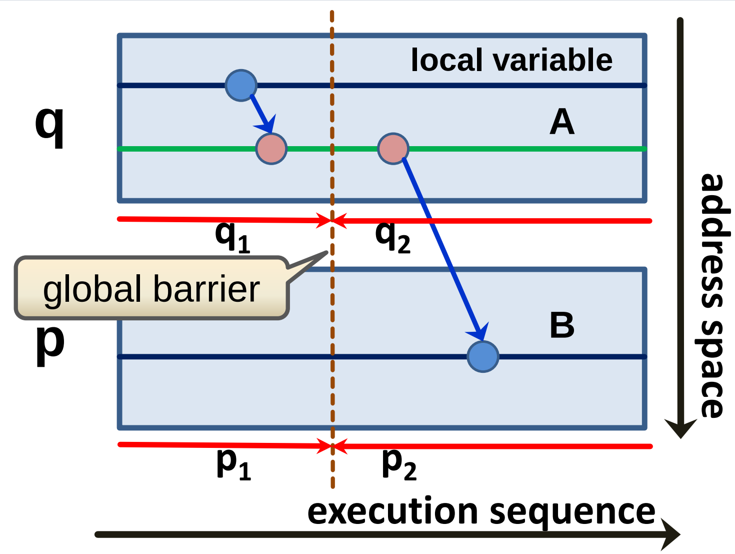


- causes race condition → **violates** language rules

- **Image control statement**

```
a = …
SYNC ALL
IF (this_image() == p) &
        b = a(:)[q]
```

programmer's responsibility



- enforce segment ordering: **q_1 before p_2**, p_1 before q_2
- q_j and p_j are **unordered**

Semantics of SYNC ALL

- **All images synchronize:**
  - SYNC ALL provides a global barrier over **all** images
  - segments preceding the barrier on any image will be ordered before segments after the barrier on any other image → implies ordering of statement execution

- **If SYNC ALL is not executed by all images,**
  - the program will discontinue execution indefinitely (**deadlock**)
  - however, it is allowed to execute the synchronization via two different SYNC ALL statements
    (for example in two different subprograms)
- **For large image count or sparse communication patterns, exclusively using SYNC ALL may be too expensive**
  - limits scalability, depending on algorithm (load imbalance!)

General synchronization rules

- **Synchronization is required**
  - between segments on **any** two different images P, Q
  - which both access the **same entity** (may be local to P or Q or another image)

  (1) P writes and Q writes, or

  (2) P writes and Q reads, or

  (3) P reads and Q writes.

- **Status of dynamic entities**
  - replace „P writes“ by „P allocates“ or „P associates“
  - will be discussed later (additional constraints exist on who is allowed to allocate)
- **Synchronization is not required**
  - for concurrent reads
  - for entities that are defined or referenced via atomic procedures

Completing the M*v: Broadcast results to all images

Assumption: must update V on each task with values from B

- **Using "Pull" implementation variant**
  - modified declaration

```
REAL :: Mat(MB, N), V(N)
REAL :: B(MB)
```

→

```
REAL :: Mat(MB, N), V(N)
REAL :: B(MB)[*]
```

only B needs to be accessible across images

  - first suggestion for communication code:

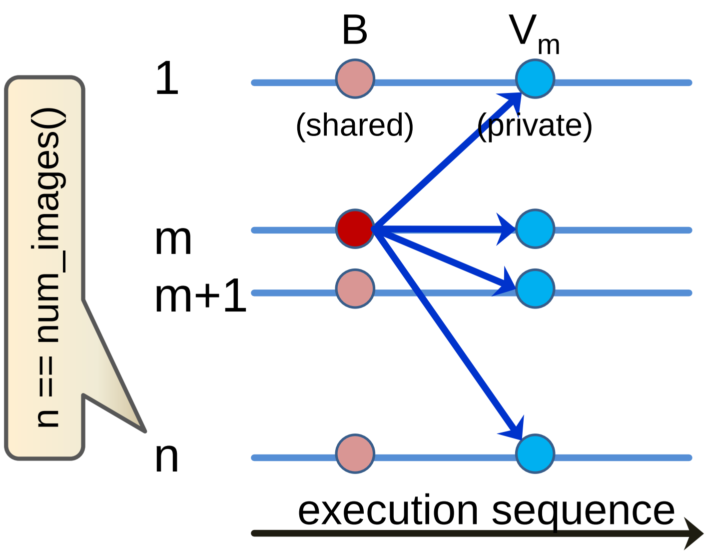
```
CALL sgemv(…)
SYNC ALL   ! assure remote B is available

DO m=1, num_images()
  V((m-1)*MB+1:m*MB) = B(:)[m]
END DO
: ! use V again
```

Formally, a correct solution ... but what about **performance?**

Optional slide

Analyzing the communication pattern

- **In m-th loop iteration:**



  - effectively, a collectively executed scatter operation
  - note that **each** image concurrently executes a communication statement

- **Slowest communication path**
  - might be a network link between two images, with bandwidth BW in units of GBytes/s
  - subscription factor is **n**
  - estimate for transfer duration of each loop iteration is

$$T = T_{lat} + \frac{MB * Size(real) * \boldsymbol{n}}{BW}$$

  (latency T_{lat} included)
  - this is **unfavourable** (an $\mathbf{n^2}$ effect when all loop iterations are accounted)

Optional slide

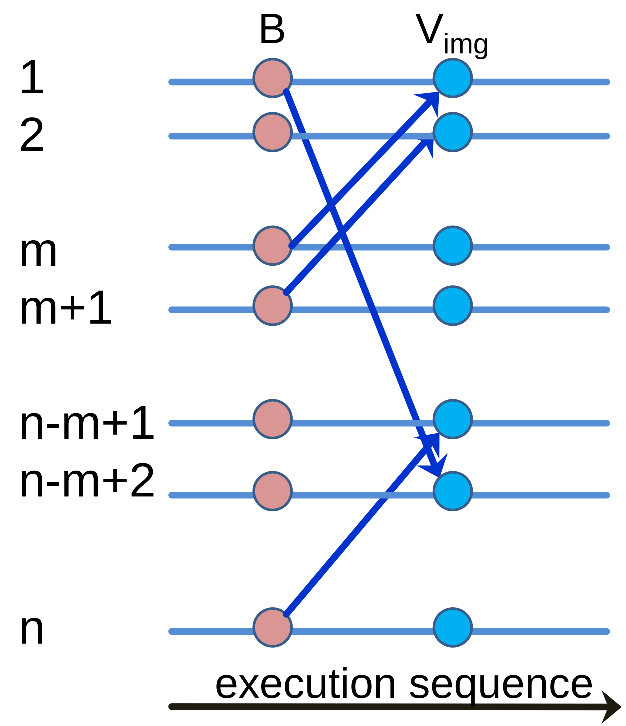
Improved communication pattern

- **Introduce a per-image shift of source image**
  - efficient pipelining of data transfer

```
CALL sgemv(…)
SYNC ALL   ! assure remote B
           ! is available
do m=1, num_images()
  img = m + this_image() - 1
  if (img > num_images()) &
      img = img - num_images()
  V((img-1)*MB+1:img*MB) = B(:)[img]
END DO
: ! use V again
```

  - balanced use of network links:

In m-th loop iteration

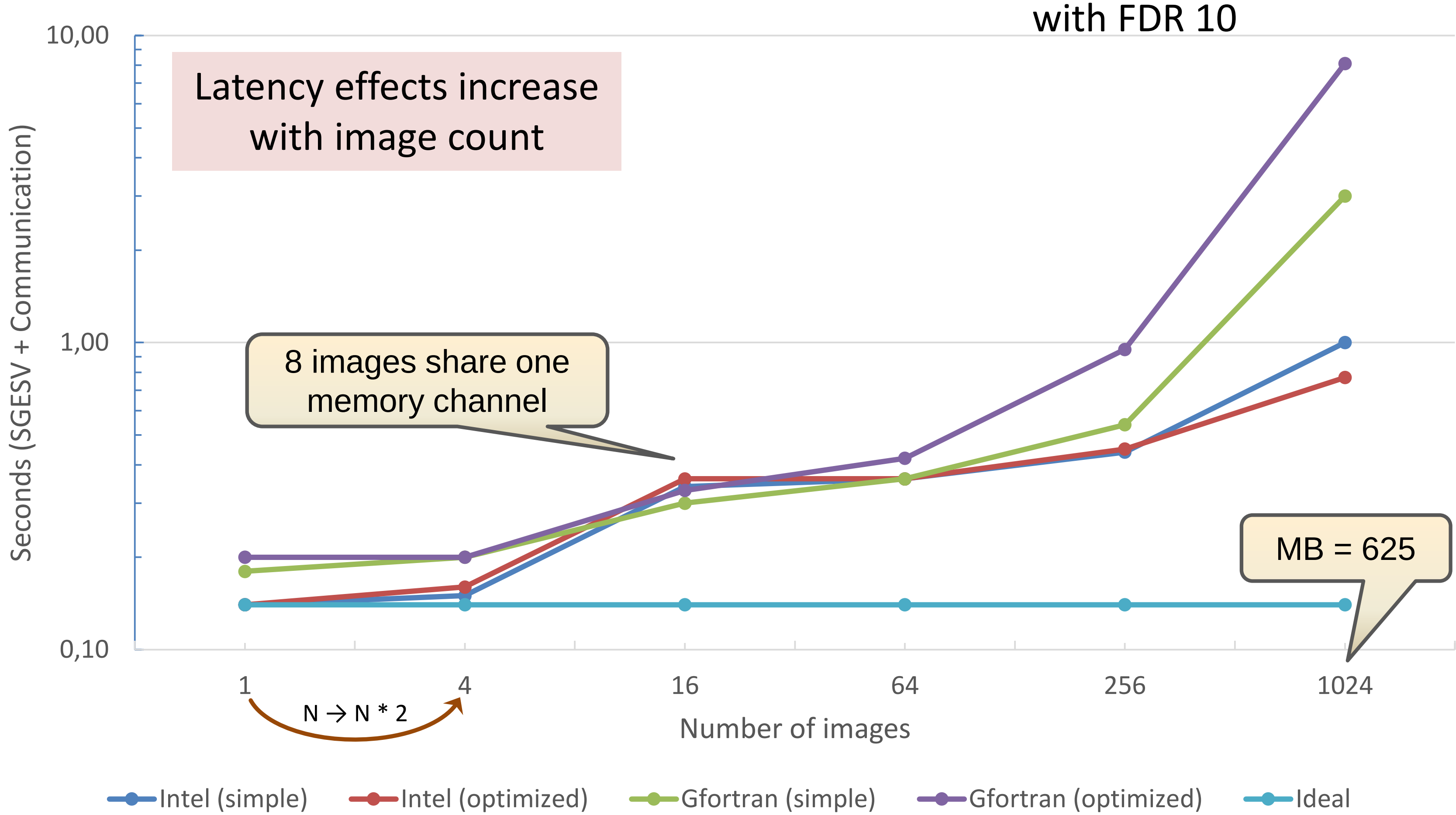


$$T \le T_{lat} + \frac{MB * Size(real) * \mathbf{(images\ per\ node)}}{BW}$$

Optional slide

Weak scaling results: $N_{(1\ image)} = 20000$

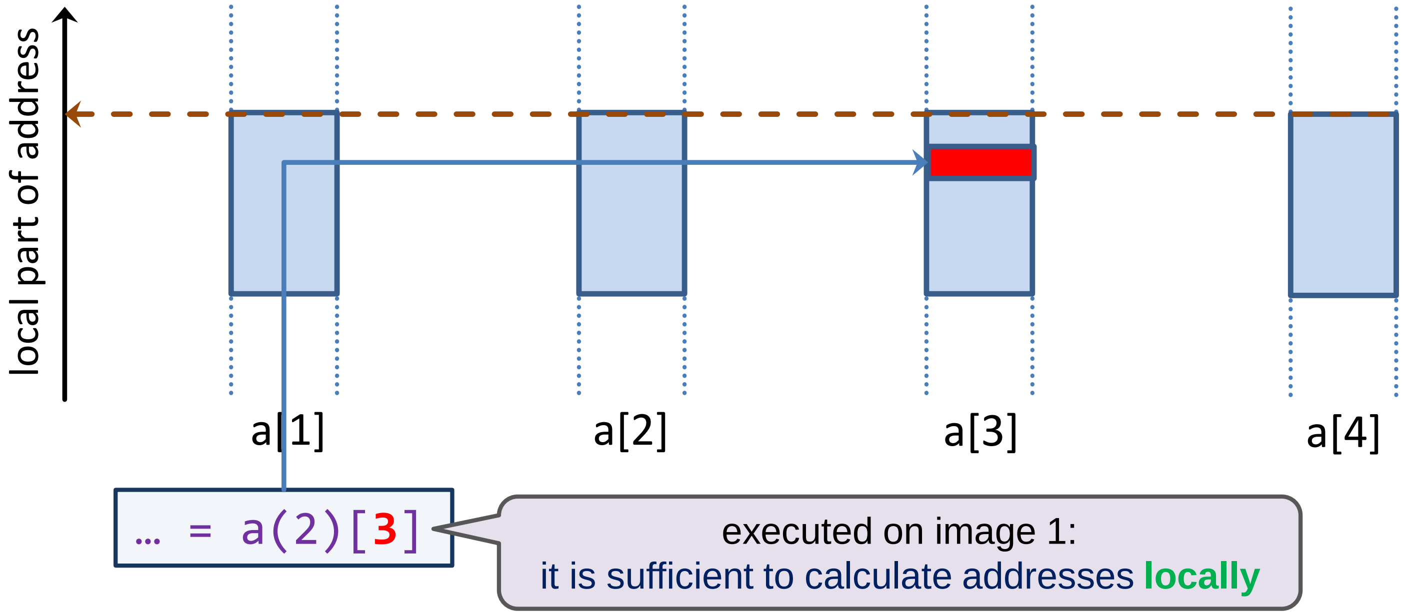
on Sandy Bridge
with FDR 10



Allocatable coarrays

Symmetric memory

- **For addressing efficiency, there is an advantage**
  - in using **symmetric** memory for coarrays (i.e. on each image, same local part of start address for a given object): no need to obtain a remote address for accessing remote elements



  - carry this property over to dynamic memory: **symmetric heap**

Declaration

a coarray **cannot** have the POINTER attribute

Allocatable object

```
INTEGER, ALLOCATABLE :: id(:)[:]
TYPE(body), ALLOCATABLE :: pavement(:,:)[:]
```

intrinsic type

derived type

both shape and coshape are **deferred**

Allocatable component

- part of type declaration

```
TYPE :: co_vector
  REAL, ALLOCATABLE :: v(:)[:]
END TYPE
```

component is an allocatable array

- objects of such a type must be **scalars**

```
TYPE(co_vector) :: a_co_vector
```

and are **not permitted** to have the ALLOCATABLE or POINTER attribute, or to themselves be coarrays

Allocatable coarrays: Executing the allocation

- **Symmetric and collective:**
  - the same ALLOCATE statement must be executed on **all images** in unordered segments

same bounds **and** cobounds
(as well as type and type parameters)
must be specified on all images

```
ALLOCATE (id(n)[0:*], pavement(n,10)[p,*], STAT=my_stat)

ALLOCATE ( a_co_vector % v(m)[*] )
```

- **Semantics:**

permits an implementation to make use of a symmetric heap

1. each image performs allocation of its **local** (equally large) portion of the coarray
2. if successful, all images **implicitly** synchronize against each other

subsequent references or definitions are
race-free against the allocation

Deallocation of allocatable coarrays

- **Symmetric and collective:**
  - the same DEALLOCATE statement must be executed on **all images** in unordered segments

```
DEALLOCATE( id, pavement, a_co_vector % v )
```

  - for objects without the SAVE attribute, DEALLOCATE will be executed **implicitly** when the object's scope is left
- **Semantics:**
  1. all images synchronize against each other
  2. each image performs deallocation of its **local** portion of the coarray

preceding references or definitions are
race-free against the allocation

Collective Procedures

Note:

Currently, these are not yet generally supported in compilers

Motivation

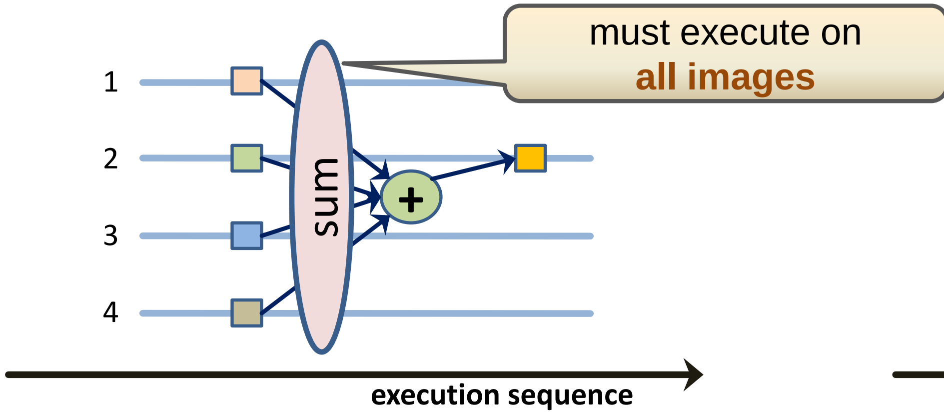
- **Common pattern in serial code:**
  - use of reduction intrinsics, for example:
    SUM for evaluation of global system properties

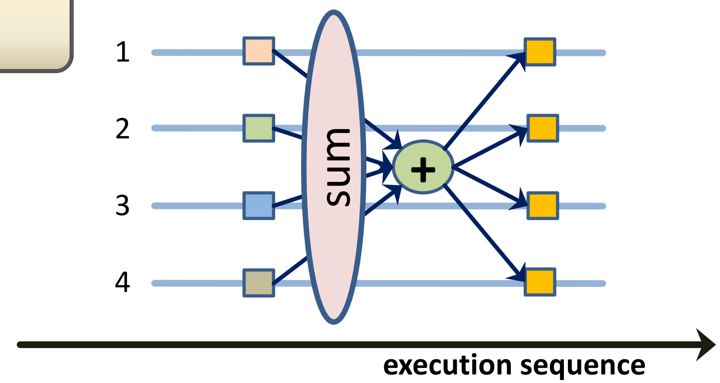
```
REAL :: mass(ndim,ndim), velocity(ndim,ndim)
REAL :: e_kin
:
e_kin = 0.5 * sum( mass * velocity**2 )
```

Quiz: what problem might arise here?

- **Coarray code:**
  - on each image, an image-dependent **partial sum** is evaluated
  - i. e. the intrinsic is not image-aware
- **Variables that need to have the same value across all images**
  - e.g. global problem sizes
  - values are initially often only known on one image

Sum reduction






```
REAL :: a(2)
:
CALL co_sum(a, result_image=2)
```

a becomes undefined on images ≠ 2

```
REAL :: a(2)
:
CALL co_sum(a)
```

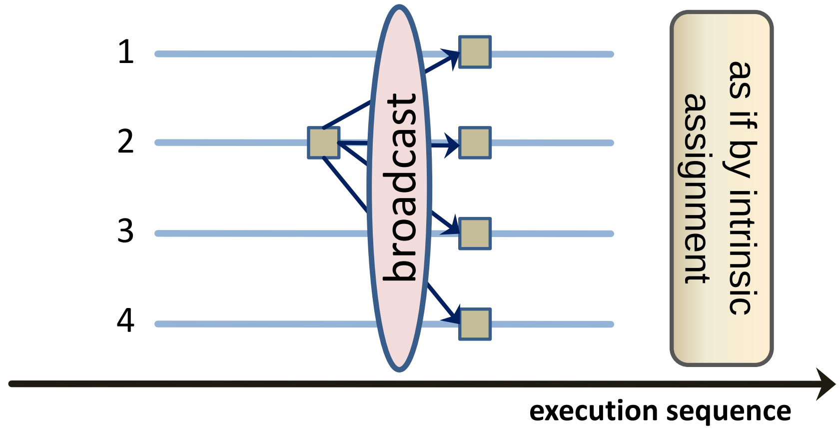
a becomes defined on all images

- **Arguments:**
  - **a** may be a scalar or array of numeric type
  - **result_image** is an optional integer with value between 1 and num_images()
  - without **result_image**, the result is broadcast to **a** on all images, otherwise only to **a** on the specified image

Further reduction procedures

- **CO_MAX**
- **CO_MIN**
- **CO_REDUCE**
  - general facility
  - permits specifying a user-defined function that operates on derived type arguments

Data redistribution with CO_BROADCAST




```
TYPE(matrix) :: xm
:
CALL co_broadcast(a=xm, source_image=2)
```

- **Arguments:**
  - `a` may be a scalar or array of any type. it must have the same type and shape on all images. It is overwritten with its value on `source_image` on all other images
  - `source_image` is an integer with value between 1 and `num_images()`

Further comments on collective subroutines

- **All collectives are "in-place"**
  - programmer needs to copy data argument if original value is still needed
- **Data arguments need not be coarrays**
  - however if a coarray is supplied, it must be the same (ultimate) coarray on all images

> For coarrays, all collectives could of course be implemented by the programmer. However it is expected that **collective subroutines will perform better**, apart from being more generic in semantics.

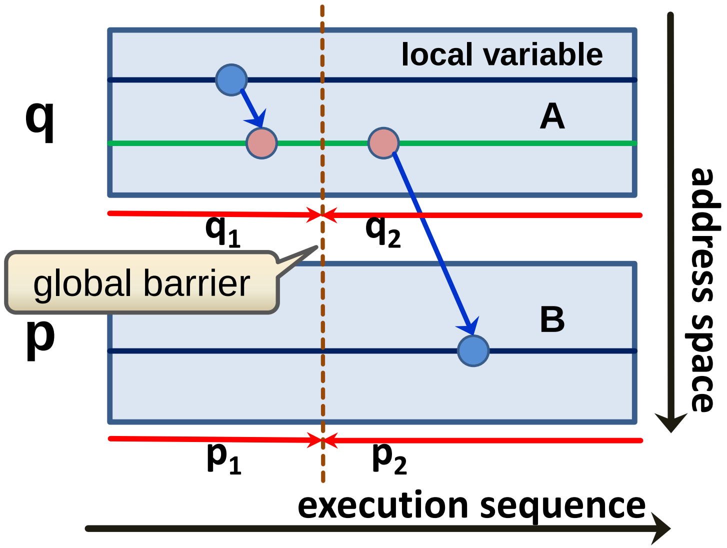
- **No segment ordering is implied by execution of a collective**
- **Collectives must be invoked by all images**
  - and from unordered segments, to avoid deadlocks

Minimal synchronization with Events

Optional slide

Weaknesses of previously treated synchronization constructs

- **Recall semantics of SYNC ALL**



  - enforces segment ordering: q_1 before p_2, p_1 before q_2
  - q_j and p_j are unordered

- **Symmetric synchronization is overkill**
  - the ordering of p_1 before q_2 is often not needed
  - image q therefore might continue without waiting
- **Therapy:**
  - TS 18508 introduces a **lightweight, one-sided** synchronization mechanism – **Events**

```
USE, INTRINSIC :: iso_fortran_env

TYPE(event_type) :: ev[*]
```

special opaque derived type; all its objects must be coarrays

Optional slide

One-sided synchronization with Events

- **Image q executes**

```
a = …
EVENT POST ( ev[p] )
```

  - and continues **without** blocking

- **Image p executes**

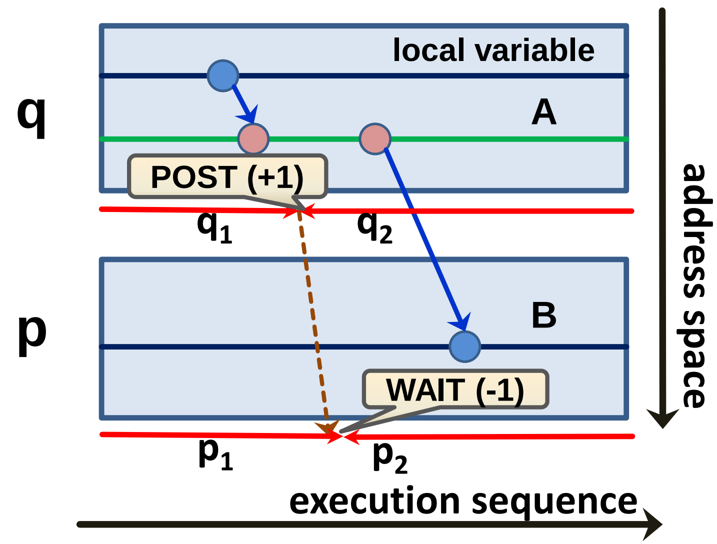
```
EVENT WAIT ( ev )
b = a(:)[q]
```

no coindex permitted on event argument here

  - the WAIT statement **blocks** until the POST has been received. Both are image control statements.

an event variable has an internal counter with default value zero; its updates are **exempt** from the segment ordering rules („atomic updates“)

- **One sided segment ordering**



  - **q_1 ordered before p_2**
  - no other ordering implied
  - no other images involved

Optional slide

The dangers of over-posting

- **Scenario:**
  - Image **p** executes

```
EVENT POST ( ev[q] )
```

  - Image **q** executes

```
EVENT WAIT ( ev )
```

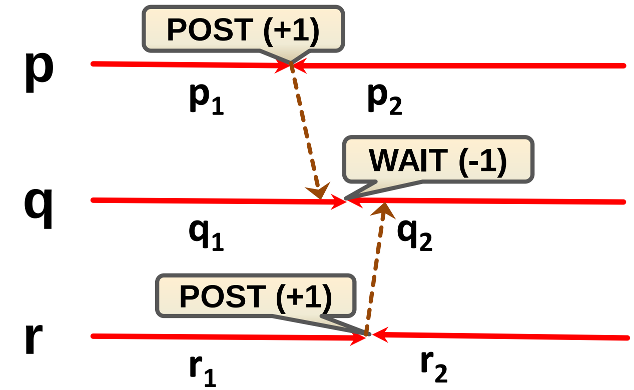
  - Image **r** executes

```
EVENT POST ( ev[q] )
```

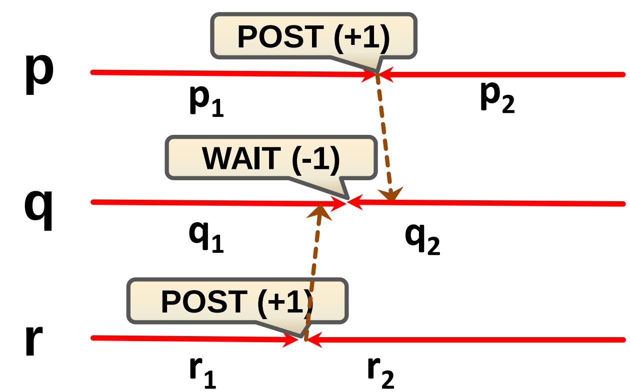
- **Question:**
  - what synchronization effect results?
- **Answer: 3 possible outcomes**
  - which one happens is **indeterminate**

Avoid over-posting from multiple images!

- **Case 1: p_1 ordered before q_2**



- **Case 2: r_1 ordered before q_2**

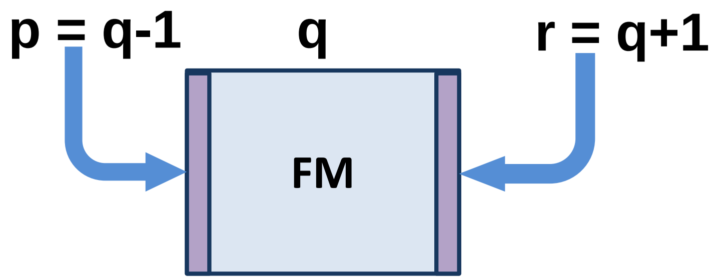


- **Case 3: ordering as given on next slide**

Optional slide

Multiple posting done correctly

- **Why multiple posting?**
  - **Example:** halo update



- **Correct execution:**
  - Image **p** executes

```
fm(:,1)[q] = …
EVENT POST ( ev[q] )
```

  - Image **r** executes

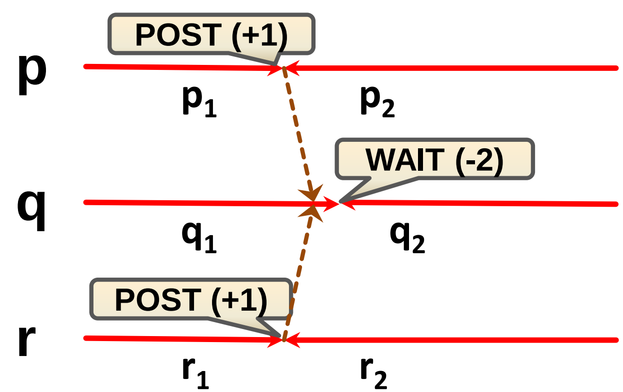
```
fm(:,n)[q] = …
EVENT POST ( ev[q] )
```

  - Image **q** executes

```
EVENT WAIT ( ev, UNTIL_COUNT = 2 )
… = fm(:,:)
```

number of posts needed

- **p_1 and r_1 ordered before q_2**



This case is enforced by using an UNTIL_COUNT

Optional slide

The EVENT_QUERY intrinsic

- **Permits to inquire the state of an event variable**

```
CALL event_query( event = ev, count = my_count )
```

  - the event argument cannot be coindexed
  - the current count of the event variable is returned
  - the facility can be used to implement non-blocking execution on the WAIT side of event processing
  - invocation has **no** synchronizing effect

Finis:

Best wishes for your future scientific programming efforts

I hope you enjoyed the event!