



An introduction to the Fortran programming language

Reinhold Bader

Nisarg Patel, Gilbert Brietzke, Ivan Pribec

Leibniz Supercomputing Centre

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
Fortran 90 (1991)	large revision
Fortran 95 (1997)	small revision
Fortran 2003 (2004)	large revision
Fortran 2008 (2010)	mid-size revision
TS 29113 (2012)	extends C interop
TS 18508 (2015)	extends parallelism
Fortran 2018 (2018)	current standard

F95

F03

F08

F18

TS → Technical Specifications

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

■ 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 (details are not covered in this course)

■ Implementation dependencies



Processor dependent behaviour (may be unportable)

■ Performance



language feature for performance

■ 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 **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**

some of the above are outside the scope of this course


■ 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

- **Original language: imperative, procedural**
 - a large fraction of original language syntax and semantics is still relevant for today
- **Fortran still supports „obsolescent“ legacy features**
 - ability to compile and run older codes
 - some are rather cumbersome to learn and use → recommend code update to modern language if it is actively developed
- **Scope of this course:**
 - a (slightly opinionated) subset of **modern** Fortran – mostly **F95** with a few features from **F18**
 - legacy features will be largely omitted (their existence might be noted)
 - content is mostly targeted at new code development

- **Modern Fortran explained** (8th edition incorporates )
 - 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**
 - W. Brainerd. Springer, 2015
- **Download of (updated) PDFs of the slides and exercise archive**
 - freely available under a creative commons license
 - <https://doku.lrz.de/display/PUBLIC/Programming+with+Fortran>



Basic Fortran Syntax

Statements, Types, Variables, Control constructs

■ First programming task:

- calculate and print the real-valued solutions of the quadratic equation

$$2x^2 - 2x - 1.5 = 0$$

- mathematical solution for the general case $ax^2 + bx + c = 0$ is

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
program solve_my_quadratic
implicit none
real, parameter :: a = 2.0, b = -2.0, c = -1.5
real :: x1, x2
intrinsic :: sqrt
:
end program
```

main program unit (exactly one needed)

enforce strong typing (best practice!)

variable (modifiable
→ needs storage space)

named constants (unmodifiable)

enforce that the function defined
in the Fortran run-time library is used

executable statements: see next slide

indenting: done
by convention

Declarative and executable statements

- **Statements on previous slide:** declarative only
 - determine entities that will be used, and their properties
- **Added statements on this slide:** will be executed when program is run

```
program solve_my_quadratic
  implicit none
  real, parameter :: a = 2.0, b = -2.0, c = -1.5
  real :: x1, x2
  intrinsic :: sqrt
```

declarations **must always precede** executable statements

assignment

intrinsic function call

```
x1 = ( -b + sqrt(b**2 - 4. * a * c) ) / ( 2. * a )
x2 = ( -b - sqrt(b**2 - 4. * a * c) ) / ( 2. * a )
```

expression

```
write(*, fmt=*) 'Solutions are: ', x1, x2
```

I/O statement
(output)

string literal

```
end program
```

Executed in order
of appearance

Compiling and running (simplest case)



Dependency:

- on processor (aka compiler) and operating system

- will produce an output like

```
Solutions are: 1.50000000 -0.50000000
```

■ For example program,

- store program text in ASCII text file
`solve_my_quadratic.f90`
- compile on Linux/UNIX:

```
ifort -o prog.exe solve_my_quadratic.f90
```

UNIX-specific note:

If the `-o` option is omitted, a `.out` is used as executable name

■ Execution of resulting binary

```
./prog.exe
```

huge numbers of additional compiler options are typically available

■ Compiled vs. interpreted code

- efficiency of execution
- typical speed factors: **20 - 60**
- greatly care for large programs



Invocations for various compilers

Vendor (Platform)	most recent version	Invocation
IBM (Power)	17.1	xlf, xlf2008, xlf2008_r
Intel (x86, x86_64)	2023.0	ifort / ifx
NVidia (x86, accelerators)	23.1	nvfortran
GCC (many)	12.2	gfortran
NAG (x86, mac)	7.1	nagfor
Cray (HPE/Cray)	13.0	ftn
ARM (arm)	22.1	armflang

LLVM-based next-generation compiler

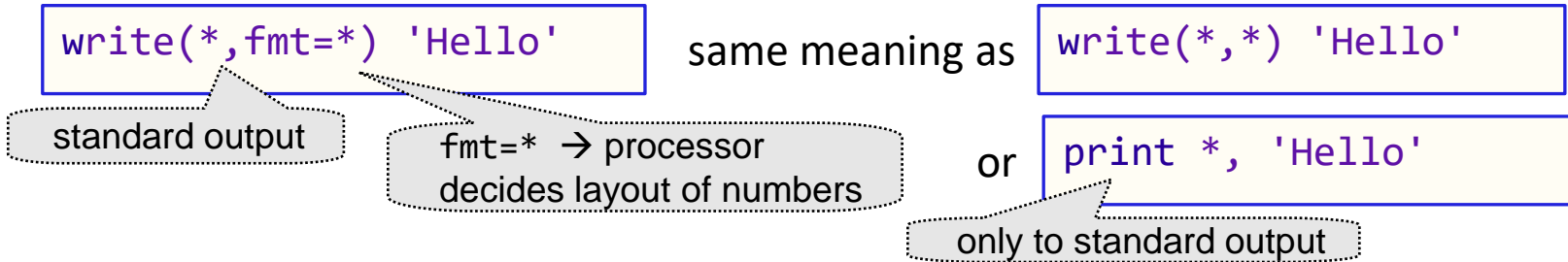
LLVM-based successor to PGI Fortran

Note:

- Operating environment usually Linux/UNIX
- On x86 or x86_64, some compilers also support the Windows or MAC operating systems
- compilers marked green are available in the hands-on sessions, possibly using a slightly older version

More on I/O

List-directed formatted output



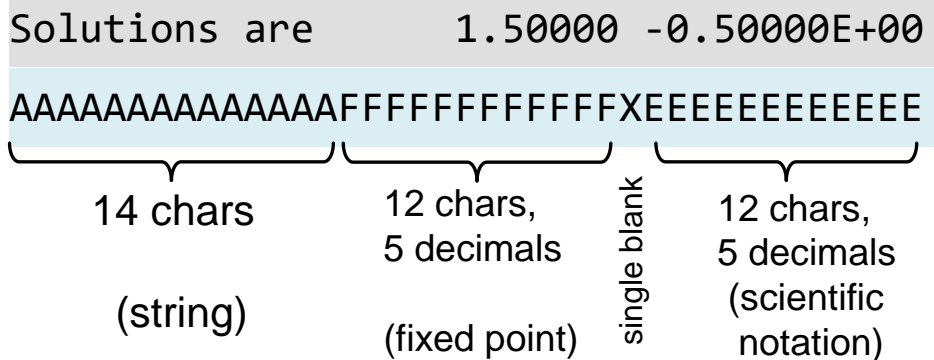
Quiz: how does one obtain data inside the program from standard input?

Programmer control over layout:

- specify an explicit format string:

```
write(*,fmt='(A,F12.5,1X,E12.5)') 'Solutions are ', x1, x2
```

will produce output



Quiz: how might the format string for integer output look like?

More on source layout

„free source form“

■ Program line

- upper limit of 132 characters

■ 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)



Legacy feature

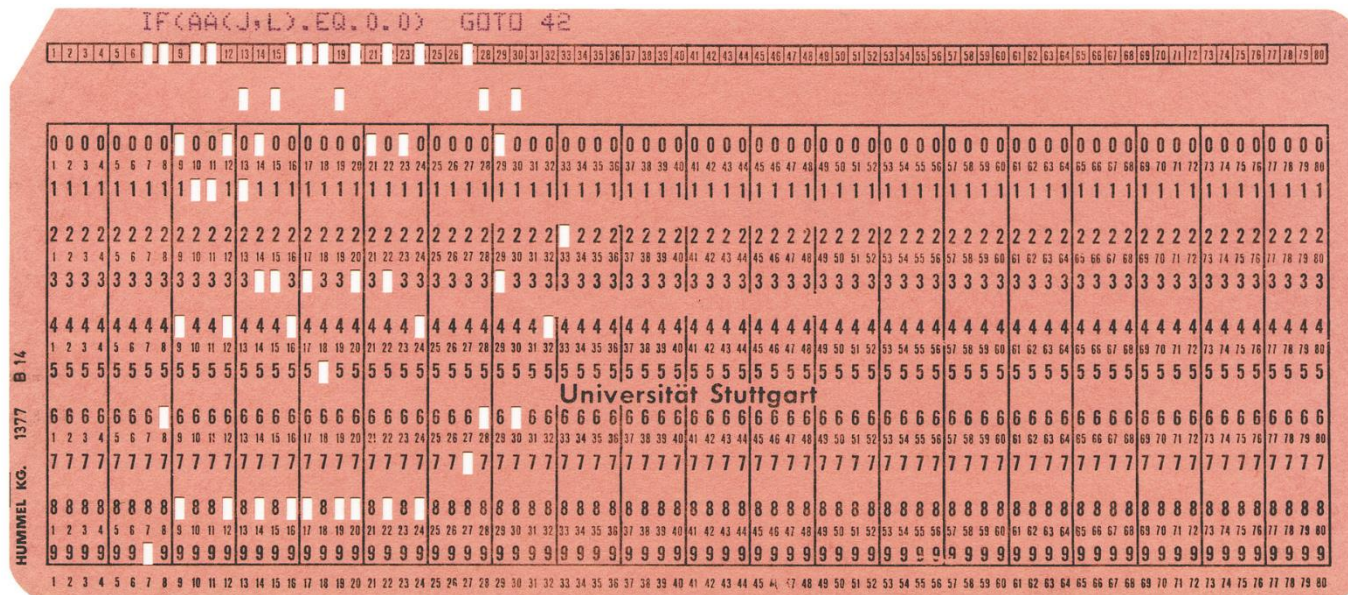
Uses file extensions:
.f .ftn .F .for

- to be avoided
- not further discussed in this course
- conversion tools exist



Technical reason:

- a relic from an earlier age of computing: the punched card



Case insensitivity

- For mostly historical reasons,

```
a = 0.0  
write(*,*) 'Hello:', a
```

means exactly the same as

```
A = 0.0  
WRITE(*,*) 'Hello:', A
```

- Mixing upper and lower case is also permitted
- However,

```
write(*,*) 'Hello'
```

```
write(*,*) 'HELLO'
```

will write two different strings to standard output

Rules for names

Names are used for referencing many Fortran entities

- e.g., variables, program units, constants, ...

Constraints:

- between **1** and **63** alphanumeric (a – z, A – Z, 0 – 9, _) characters
- first character **must** be a letter

<p>k_reverse q123 Xx</p>	<p>legal</p>	<p>1_fish a fish! \$sign</p>	<p>non-conforming</p>
<p>same reference as xx or XX</p>			

Recommendations for naming:

- no reserved words in Fortran → but do **not** use keywords anyway
- mnemonic names → relationship to use of variable

integer :: day, month, year

■ Recommendation: Enforce **strong** typing

`implicit none`

→ all object's types must be declared

 avoid legacy implicit typing

■ Three **numeric** intrinsic types

numeric storage unit: typical value nowadays **4 bytes**

1. integer
2. real
3. complex

```
integer :: years  
real    :: mass  
complex :: psi
```

one storage unit each

two storage units

■ Two **non-numeric** intrinsic types

4. character
5. logical

```
character :: c = 's'  
logical  :: flag = .true.
```

single character

or `.false.`

initialization of variable

■ Non-intrinsic types

- derived types will be discussed later

■ An object declared `integer`

- can only represent values that are a **subset** of $Z = \{0, \pm 1, \pm 2, \dots\}$
- typically $\{-2^{31} + 1, \dots, +2^{31} - 1\}$
- may be insufficient in some cases

$2^{32} - 1 \rightarrow 2,147,483,647$

■ **KIND** type parameter

- used for non-default representations:

minimal decimal exponent

```
integer, parameter :: lk = selected_int_kind(16)
```

value is not portable

two storage units

```
integer(kind=lk) :: seconds = 31208792336_lk
```

Type parameters (2)

■ An object declared `real` (or `complex`)

- can only represent values that are a **subset** of the real (or complex) field

■ **KIND** type parameter

- used e.g. for non-default representations

```
integer, parameter :: dk = selected_real_kind(13,200)
```



value is not portable

```
real(kind=dk) :: charge = 4.16665554456e-47_dk
```

two storage units

decimal point

exponent

minimal decimal digits and exponent

- equivalent with `real(kind=dk)` declaration and initialization:

```
double precision :: charge = 4.16665554456d-47
```

Overview of supported KINDs

Integer and Real types:

- at least two KINDs must be supported
- intrinsic functions that produce KIND numbers:
selected_int_kind(), selected_real_kind(), kind()

see next slide

Real types only

- usually, KINDs for smaller exponents also exist (reduced storage requirement)
- some processors support 10 or 16 byte reals (performance may be very low)

Unsupported digit/exponent specification

- will fail to compile

IEEE defined

integer kind	max. exponent
default	10^9 ($2^{31}-1$)
extended	10^{19} ($2^{63}-1$)

real kind	dec. digits	exponent range
default	6	$10^{-37} - 10^{+38}$
extended	15	$10^{-307} - 10^{+308}$

■ Declaration:

```
complex :: c  
complex( kind=kind(1.0d0) ) :: z
```

default real

double precision

- real and imaginary part have the same KIND number
- intrinsic function `kind()` produces the KIND number of its argument

■ Complex literal constants: $(a, b) = a + ib$ (mathematical notation)

```
c = ( 1.2, 4.5e1 )
```

```
z = ( 4.0_dk, 3.0_dk )
```

where `dk` has the value `kind(1.0d0)`

■ Literal string constant

- of default kind:

```
'Full House'  
"Full House"
```

length
is 10

- single or double quotes possible; they are delimiters only and not part of the constant
- blanks and case are significant:

```
'full House'  
'FullHouse'
```

different
from above

- characters other than the Fortran set are allowed. E.g., a quoted **&** would be part of the string entity

■ Quotes as part of a character entity:

- either use the one not used as delimiter

```
'"Thanks", he said'  
" 'Thanks', he said"
```

- or use double delimiter to mask a single one:

```
'It''s true'
```

value is:
It's true

Note: no statements on this slide, tokens only

■ String variables

- require **length** parametrization

```
character(len=12) :: fh
:
fh = 'Full House'
```

because default length is one.

- auto-padded with blanks at the end (here: 2 blanks)

■ KIND type parameter

- differentiate between different character sets, for example

1. default character set
2. character set used in C
3. UTF-8 character set

In practice,

- 1. and 2. are usually the same
- will not discuss 3.

```
integer, parameter :: &
    ck = kind('A')

character(kind=ck, &
    len=12) :: fh

:
fh = ck_'Full House'
```

- special exception: character KIND number **precedes** string constant

Arrays (1) - Simple array declaration

■ Aim:


- Facilitate declaration of objects capable of holding multiple entities of a given type

■ DIMENSION attribute:

```
integer, parameter :: dm = 6  
real, dimension(dm) :: a
```

■ Alternative declaration variants:

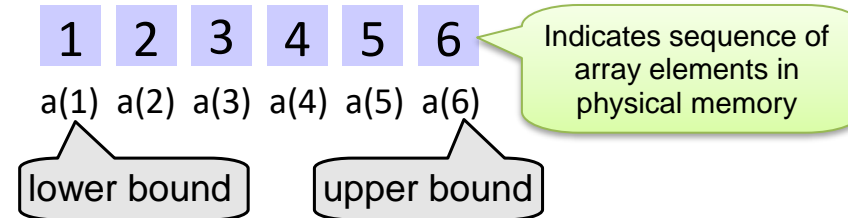
(1) `real :: a(dm)` attribute is implicit

 (2) `real :: a`
`dimension :: a(dm)` statement form

Recommendation:

avoid statement form (can be confusing if non-local)

■ Layout of (scalar) array elements in memory:



trivial mapping between storage sequence index and array index applies only for simple arrays

Arrays (2): How to use simple arrays

■ Array construction:

- With a single statement

```
a = [ 1.,3.,5.,7.,9.,11. ]
```

- Legacy notation (equivalent)

```
a = (/ 1.,3.,5.,7.,9.,11. /)
```

- Value of array elements after execution of above statement:

1.	3.	5.	7.	9.	11.
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)

■ References and definitions of array elements: subscripting

```
integer :: i  
real :: t1  
i = 2  
a(3) = 2.0  
t1 = a(3)  
a(i) = t1*3.0  
t1 = t1 + a(i+4)
```

Scalar integer subscript:
• **constant**
• **variable**
• **expression**

1.	6.	2.	7.	9.	11.
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)

- t1 will have value **13.**

- the above addresses **small, simple** arrays or **single** array elements
- mechanisms to process complete/large arrays are needed
- there's **much** more to array support in Fortran than this – stay tuned

Conditional execution (1)

■ Argument of `sqrt()`

- a non-negative real number is required („discriminant“)
- to avoid non-conforming code, replace executable statements by

declared as `real :: disc`

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

scalar logical expression

```
if (disc >= 0.0) then
```

```
  x1 = ( -b + sqrt(disc) ) / ( 2. * a )
```

```
  x2 = ( -b - sqrt(disc) ) / ( 2. * a )
```

```
  write(*,*) 'Solutions are: ', x1, x2
```

```
else
```

```
  write(*,*) 'No real-valued solution exists'
```

```
end if
```

a block with
three statements

Conditional execution (2)

■ Repeated `else if` blocks:

```
if (scalar-logical-expr) then
  block
else if (scalar-logical-expr) then
  block
else if ... ! further blocks
:
else      ! optional
  block
end if
```

- the **first** block for which its condition is true is executed
- if none, the else block is executed, if it exists

■ IF statement:

not a block construct

```
if (scalar-logical-expr) &
  action-stmt
```

- action statement: essentially any single statement
- examples:

```
if (x < 2.) y = y + x
if (z /= 0.) a = b/z
```

„not equal“



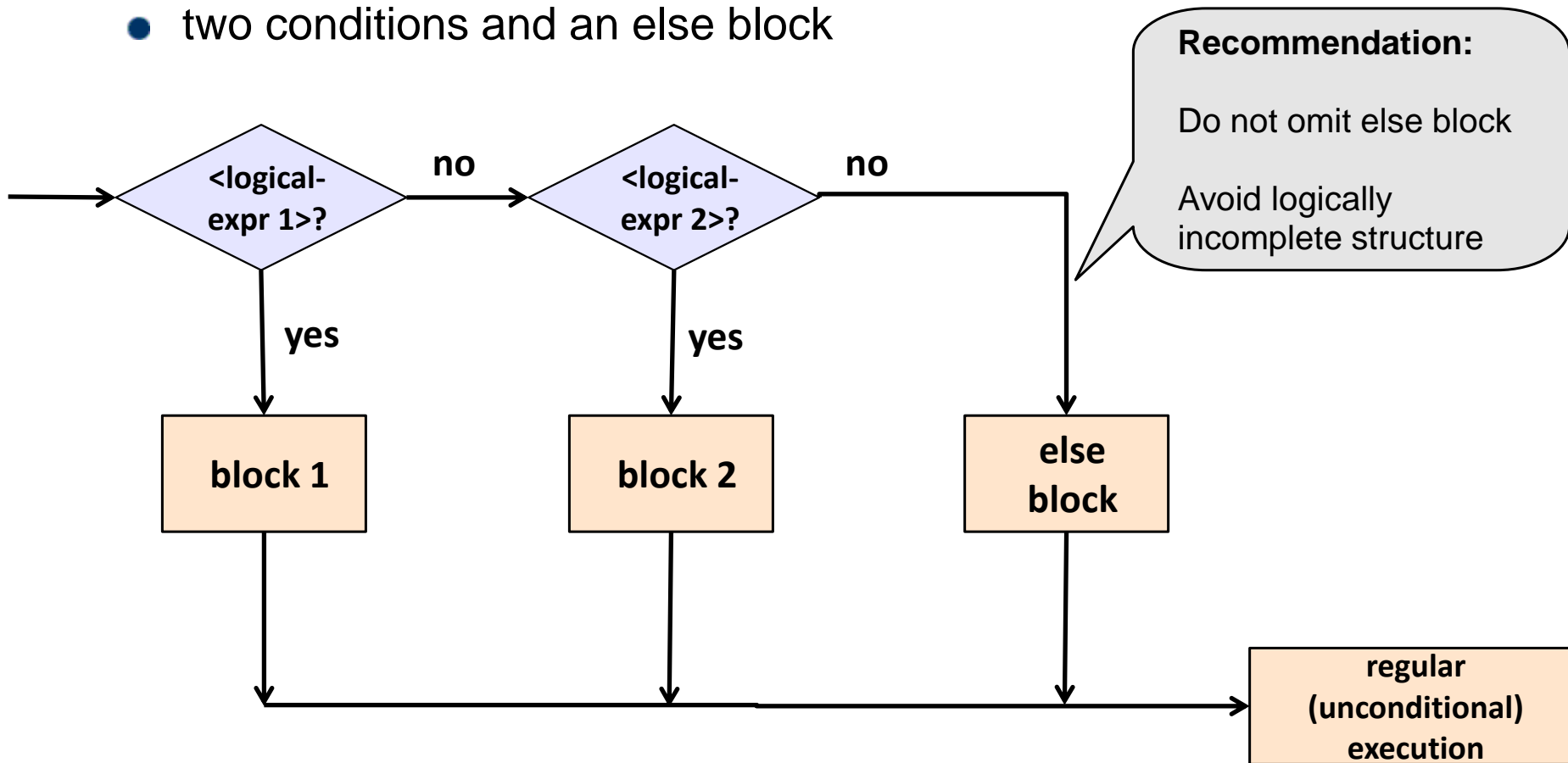
legacy form of IF:

- arithmetic if
- not discussed here

Flow diagram for conditional execution

■ Shown here:

- two conditions and an else block



Conditional execution (3)



The **CASE** construct – an alternative to multiple IF blocks:

- if only a single scalar expression of type **integer**, **character** or **logical** is evaluated
- if all values are from a pre-defined set

```
[name :] select case (expr)  
case selector [name]  
    block  
: ! possibly repeated  
end select [name]
```

■ Example:

```
select case (index)  
case (0)  
    x = 0.  
case (1:4)  
    x = 1.0  
case (5:)  
    x = 2.0  
case default  
    x = -1.0  
end select
```

here an **integer**

single value

selector

lower and upper limits

lower limit only


no case fits
(one block only)

- no overlap is allowed within or between selectors → at most one block is executed







Overview of block constructs

■ General concept:

- construct by default has one entry and one exit point
- modifies statement execution order

Labeled GOTO statements  should not be used any more

■ Overview of constructs defined in the standard:

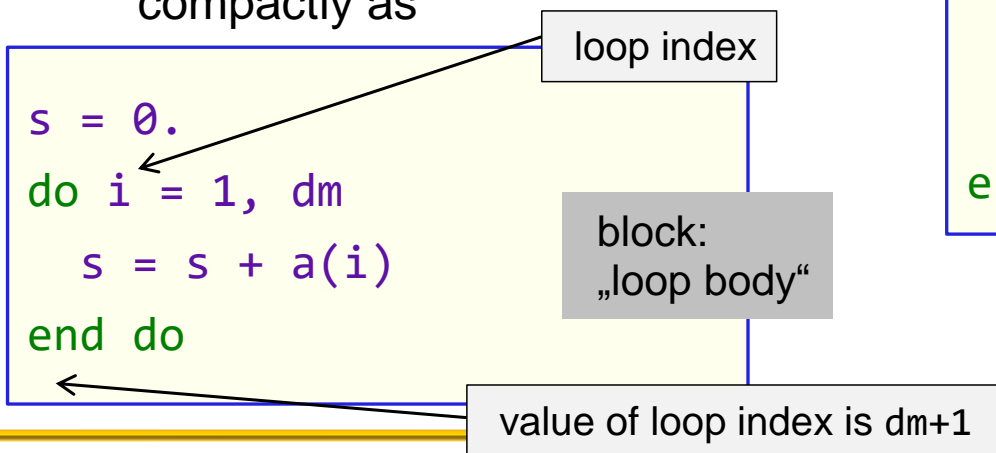
Name	Purpose		treated in this course?
ASSOCIATE	define and use block-delimited alias		no
BLOCK	define structured block, possibly with local variables		yes
CHANGE TEAM	split coarray execution into disjoint teams		no
CRITICAL	limit coarray execution to a single image		no
DO	looping construct (repeated execution)		yes
IF	conditional execution		yes
SELECT CASE	conditional execution based on predefined values		yes
SELECT RANK	run time rank resolution		yes
SELECT TYPE	run time type / class identification		no

■ Example:

- summing up the elements of an array

```
s = a(1)
s = s + a(2)
:
s = s + a(6)
```

- can be written more compactly as



■ Rules for DO constructs

- loop index **must** be an integer



loop index may **not** be modified inside the loop body (but may be referenced)

- loop index takes every value between lower and upper limit in order

■ Most general form:

```
[name:] do [,] var = &
           e1, e2 [, e3]
           body
end do [name]
```

- e1, e2, e3 must be integer **expressions**. If present, e3 must be ≠ 0.

Repeated execution (2)

Index set for general DO construct:

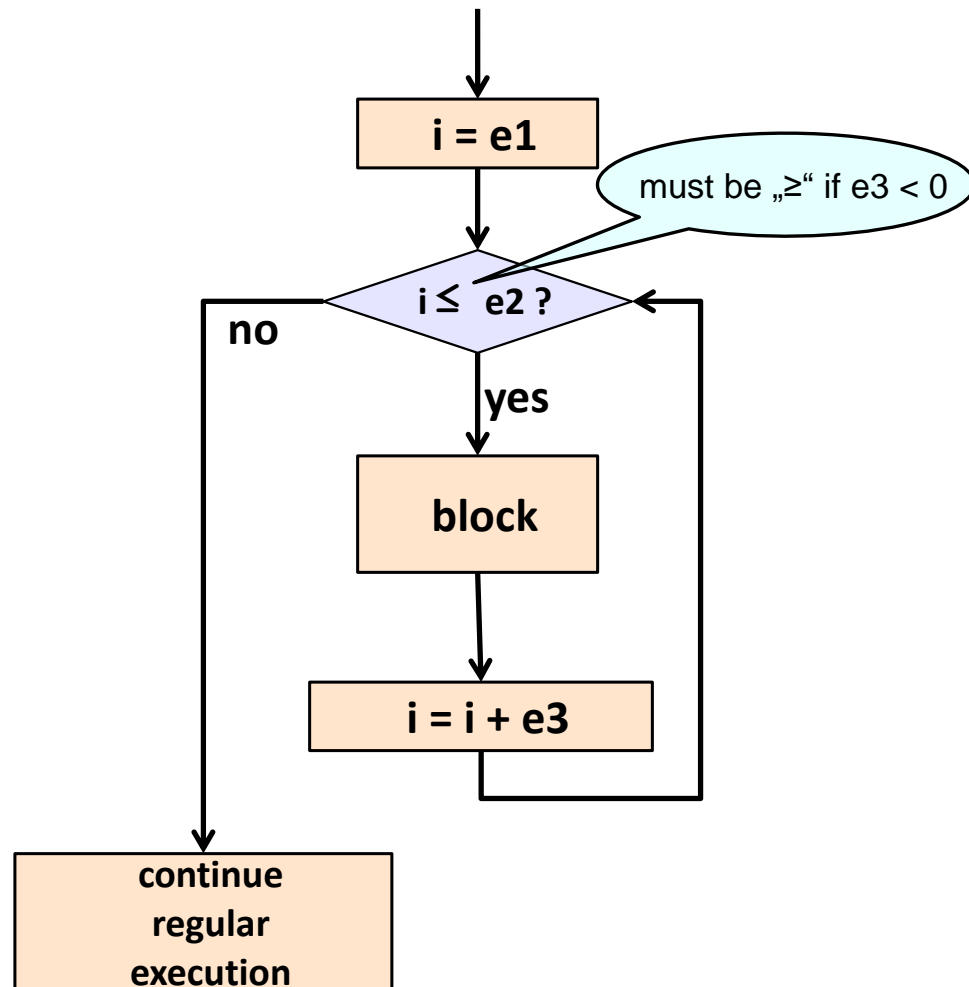
- if e_3 is not specified, set e_3 to 1
- start with e_1 and increment by e_3 as long as e_2 not exceeded
- Empty index set: loop body is **not** executed; control of execution is transferred to statement after end of loop



legacy DO:

- with labeled ending statement
- is not discussed here

Block diagram



■ Terminate a construct

- use an EXIT statement

```
i_loop: do i=1,n
:
! terminate if singular
if (den==0.0) exit i_loop
:
end do i_loop
```

- if executed, next statement is the first one after the referenced block
- if no block name is specified, applies to innermost enclosing DO block Recommendation: **avoid** this
- Applies for **all** block constructs

■ Proceed to next loop iteration

- Use a CYCLE statement

```
j_loop: do j=1,n
:
! Next iteration if negative
if (t<0.) cycle j_loop
a(j) = a(j) + sqrt(t)*...
end do j_loop
```

Actually, $\theta(t) \cdot \sqrt{abs(t)}$

- if executed, restart referenced loop body with the next value for the iteration variable
 - if no block name is specified, applies to innermost enclosing DO block
- ## ■ Only for looping constructs

■ Endless DO construct

```
[ name: ] do
  :
  if (scalar-logical-expr) exit
  :
end do [ name ]
```

- requires a conditioned **exit** statement to eventually complete

■ DO WHILE construct

```
[ name: ] do while ( .not. scalar-logical-expr )
  :
end do [ name ]
```

condition from above

- condition is checked **before** block executed for each iteration
- equivalent to previously shown „endless“ DO with conditional branch as its **first** block statement



use not recommended since not well-optimizable

■ Semantics:

- delineated block of executable statements, which are executed in the usual order, beginning with the first one appearing inside the construct
- **optionally**, prepended by block-local variable declarations – these variables only exist while the block executes

This is not permitted for the other block constructs

- **optionally**, the block construct may be given a name
- an **exit** statement can appear as one of the executable statements. If it references the given construct (e.g., by name), execution continues after the block

■ Example:

```
real :: result
: ! executable statements

do_some_task : block
integer :: i
real :: b(n)

:
b(i) = ...
:
result = ...

end block do_some_task

: ! executable statements
... = result + ...
```

construct name

local declarations

executable statements

local **b,i** cease to exist



result remains available

Nesting of block constructs and fine-grain execution control

■ Nesting is permitted

- A complete construct inside another one

■ Example 1: nested loops

```
outer: do j=1,n
  do k=1,n
    : 
    if (t < 0.0) cycle outer
  end do
  a(j) = a(j) + sqrt(t)*...
end do outer
: 
```

- Using a name for the DO construct is necessary here
- EXIT (on the inner loop) would not be sufficient here

■ Example 2:

- loop nested inside a BLOCK construct
- IF nested inside loop

```
ifound = 0
finder : block
  integer :: i
  do i=1,n
    if (x == a(i)) then
      ifound = i
      exit finder
    end if
  end do
  write(*,*) 'Not found'
end block finder
:
```

■ Syntax alternatives:

F08

stop

error stop

stop <integer-constant>

error stop <integer-constant>

stop <string-constant>

error stop <string-constant>

■ Semantics:

- stops execution of the complete program
- 🔍 provided **access code** is usually printed to error output
- 🔍 an integer constant may also be propagated as process exit value
- for **serial** programs, no substantive difference between the two (for parallel programs that e.g. use coarrays, there is a difference)



Model numbers, Expressions and Assignment

Numeric models for integer and real data

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

integer **kind** is defined by

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

integer **value** is defined by

- sign $s \in \{\pm 1\}$
- sequence of $w_k \in \{0, \dots, r-1\}$

base 2 → „Bit Pattern“

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

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 $f_k \in \{0, \dots, b-1\}$,
 f_1 nonzero

Inquiry intrinsics for model parameters

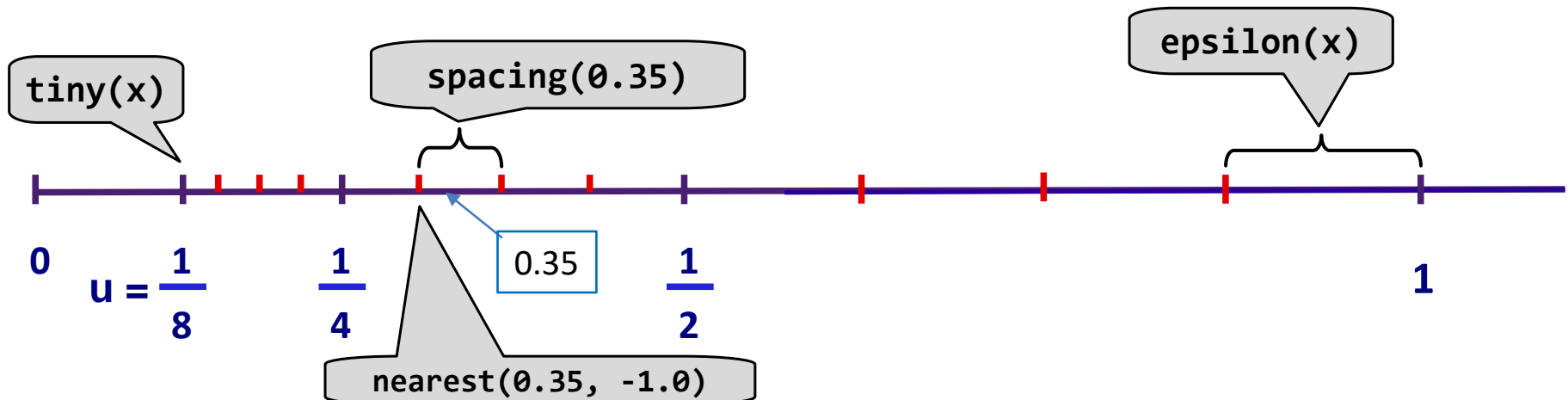
digits(x)	for real oder integer x, returns the number of digits (p, q respectively) as a default integer value.	minexponent(x), maxexponent(x)	for real x, returns the default integer e_{min}, e_{max} respectively
precision(x)	for real or complex x, returns the default integer indicating the decimal precision (=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: $e \in \{-2, -1, 0, 1, 2\}$, $p=3$

purely illustrative!

- look at first positive numbers (spacings $\frac{1}{32}, \frac{1}{16}, \frac{1}{8}$ etc.)



$$\text{rrspacing}(x) = \text{abs}(x) / \text{spacing}(x)$$

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

huge(x)

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

- to nearest model number
- maximum relative error

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

- **Typically used representations: IEEE-754 conforming**
 - matched to hardware capabilities

real kind	dec. digits	base 2 digits	dec. exponent range	base 2 exponent range
default	6	24	$10^{-37} \dots 10^{+38}$	-125 ... +128
extended	15	53	$10^{-307} \dots 10^{+308}$	-1021 ... +1024

- **Negative zero:**

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

■ Additional numbers outside model may exist

■ IEEE-754 adds

- denormal numbers (minimal exponent and $f_1=0$), decreasing precision
- infinities (Inf)
- not a number (NaN)
- register values with increased range and precision

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

■ Arithmetic operations:

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

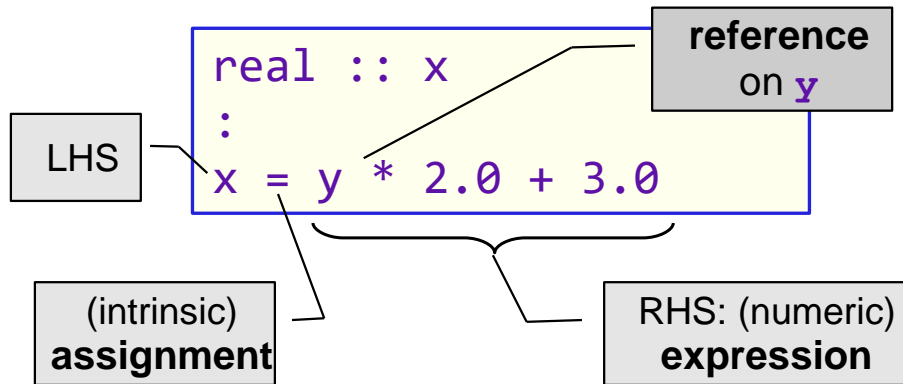
$$fl_{op}(x, y) = (x \ op \ y) \cdot (1 + d),$$
$$|d| \leq u; \ op = +, -, *, /.$$

- precision achieved by using e.g., guard digits

■ IEEE-754 adds

- more rounding functionality
- fulfills the standard req. above

Simple example



Exact semantics:

1. value of **expression** on RHS is evaluated (stay tuned for rules on this)
2. if possible (and necessary), **conversion** to the type of the LHS is performed
3. the LHS takes the previously evaluated value (it becomes **defined**)

Rationale: enable safe execution of

```
x = y * 2.0 + x * 3.0
```

assumption: has been previously defined

Notes:

- these semantics apply for all intrinsic types
- conversion is essentially limited to within numeric types. Otherwise, types and kinds of LHS and RHS must be the same
- the LHS of an assignment statement must be a **definable** entity (e.g., it must not be an expression, or a named constant)

Variant 1:

- LHS an array, RHS a scalar

```
real :: a(dm)
real :: y
:
y = 4.0
a = y * 2.1
```

→ RHS is broadcast to all array elements of LHS

Variant 2:

- LHS and RHS an array

```
real :: a(dm), b(dm), c(dm+4)
a = c      ! non-conformable
a = b      ! OK
a = c(1:dm) ! OK
```

subobject of c

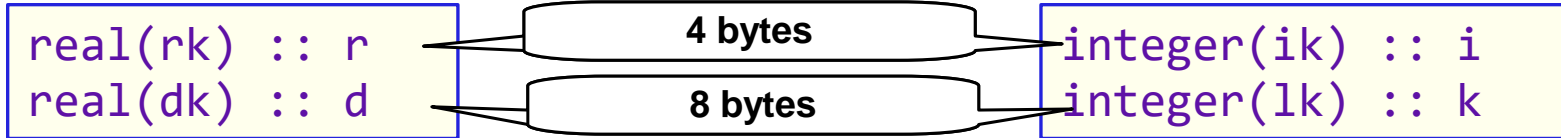
- **in this example:** of same size
→ causes element-wise copy

- **Later talks on array processing**
 - will provide more details



Implicit conversions

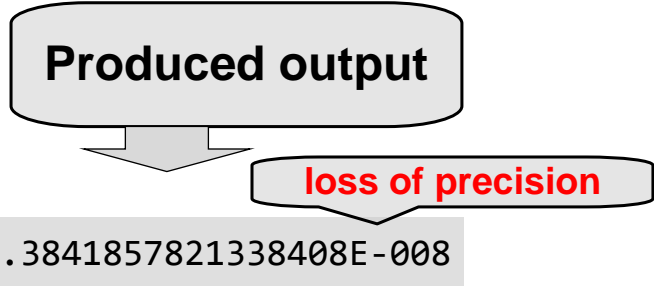
Assume declarations



Examples:

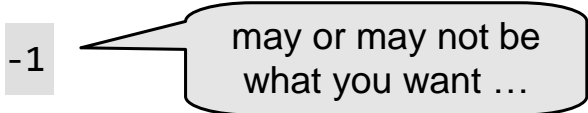
- Not exactly representable values

```
r = 1.1
d = r
write(*,*) abs(d - 1.1_dk)
```



- Rounding toward zero for real-to-integer conversion

```
r = -1.6
i = r; write(*,*) i
```



- Overflow (likely silent 🌋)

```
k = 12345678900_lk
i = k; write(*,*) i
```



- i. Use suitable intrinsics
- ii. Limit conversion to the case stronger → weaker type

- if the reverse is not avoidable, i. may help (if for clarity only)

Improved examples:

- 1. Not exactly representable values

```
d = 1.1_dk  
r = real(d, kind(r))  
write(*,*) abs(r - 1.1)
```

avoid lower-precision constants

might not be zero, but there are no unrealistic expectations

- 2. Suitable intrinsic for real-to-integer conversion

```
r = -1.6  
i = nint(r); write(*,*) i
```

-2

want rounding to nearest

- 3. Avoid overflow

```
if (abs(k) <= huge(i)) then  
  i = k  
else  
  : ! handle error  
end if
```

this also works for integers!

<code>cmplx(x [, y] [, kind])</code>	conversion to complex, or between complex KINDs
<code>int(x [, kind])</code>	conversion to integers, or between integer KINDs
<code>real(x [, kind])</code>	conversion to reals, or between real KINDs

■ Lots of further intrinsics exist, for example

<code>ceiling(a [, kind])</code> <code>floor(a [, kind])</code>	produces nearest higher (or lower) integer from real
<code>nint(a, [, kind])</code>	produces nearest integer from real
<code>anint(a, [, kind])</code>	produces nearest whole real from real

- Some of these perform conversions as part of their semantics
- KIND argument determines KIND of result
- Consult, for example, the gfortran intrinsics documentation

<https://gcc.gnu.org/onlinedocs/gfortran/Intrinsic-Procedures.html>

■ Operands and operators:

- dyadic (or binary) operators:

<operand> operator <operand>

Example tokens:

a + b

- monadic (or unary) operators:

operator <operand>

-c

■ Combining binary and unary operators: In

<operand> operator_1 operator_2 <operand>

***operator_2* must be a unary operator**

■ Operands may be

- constants
- variables
- function values
- expressions → recursively
build up complex expressions

■ Operators may be

- intrinsic operators
(depend on operand type)
- defined operators
(treated later)

■ Validity of expressions

- operands must have a well-defined value
- mathematical rules – e.g., no non-integer exponents of negative numbers
- limitations of model numbers may cause trouble sometimes

■ Initially, only operands of intrinsic types will be discussed

- note however that even intrinsic operators can be **overloaded** for derived type operands (treated later)

Expressions (3): Intrinsic numeric operators

Symbol	Meaning	Remarks
+	addition	also unary
-	subtraction	also unary
*	multiplication	
/	division	slow
**	exponentiation	even slower

Properties:

- precedence **increases** monotonically going down the table
- +, - and *, / have same precedence
- equal precedence: expression is evaluated left-to-right, **except** for exponentiation **

Some special cases:

- integer division truncates toward zero

6/3
8/3
-8/3

expression's value:

2
2
-2

- exponentiation with complex base: **a**b** produces principal value

$$e^{b \cdot (\log|a| + i \arg(a))}$$

with

$$-\pi < \arg(a) \leq \pi$$

Some examples for expression evaluation

(a, b, c, d of same numeric type and kind)

- Force order of evaluation by bracketing:

$$(a / b) / c$$

starts by evaluating a/b.
Note that

$$a / b / c$$

may be evaluated **by the processor** as

$$a / (b * c)$$

(the latter will usually be faster)

the general precedence and bracketing rules also apply for non-numeric operators

- By the precedence rules,

$$a + b * c ** d / e$$

is evaluated as

$$a + ((b * (c ** d)) / e)$$

- Equal precedence:

$$-a + b - c$$

is evaluated as

$$((-a) + b) - c$$

but (exceptionally)

$$a ** b ** c$$

is evaluated as

$$a ** (b ** c)$$

■ Operands of **same** type and kind

- expression retains type and kind

■ Operands of **differing** kinds and types

- simpler/weaker type and/or kind is coerced to the stronger type and/or kind
- then operation is performed
- result is also that of the stronger type or kind

■ Operands of **same** type but **differing** kind

- a real argument of the lower precision kind will be coerced to the higher precision kind

this does not imply higher precision of the operand's value!

- an integer argument with smaller range will be coerced to a kind that has higher range



Note: Conversion overhead can impact performance, but the extent of this is implementation-dependent

■ for $a \text{ op } b$

- with op one of the intrinsic numeric operations

Type of a	Type of b	Coercion performed
I	R	a to R
I	C	a to C
R	I	b to R, except **
R	C	a to C
C	I	b to C, except **
C	R	b to C

Legend:

I → integer

R → real

C → complex

■ Special rules for exponentiation:

- integer exponents are retained
- the compiler might convert these for improved performance:



Expressions (6): Logical operations

Operands:

- variables as well as evaluated result are of type logical

Precedence increases (.neqv. and .eqv. have same level)

a	b	a .neqv. b	a .eqv. b	a .or. b	a .and. b	.not. a
T	T	F	T	T	T	F
F	F	F	T	F	F	T
T	F	T	F	T	F	

unary

Examples:

```
logical :: a, b, c, d
: ! define a, b, c
d = ( a .or. b ) .and. c
write(*,*) d
d = a .or. .not. c
write(*,*) d
```


Operands:

- numeric or character expressions
- state truthfulness of the relation between operands → result is a **logical** value

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

- precedence: lower than numeric operators, higher than logical operators

- for complex arguments: only ==, /= allowed
- character entities: see later

Example:

```
logical :: r1, r2
real :: a
integer :: i, j
: ! define a, i, j
r1 = a >= 2.0
r2 = a < i - j
```

mixed mode expression:
coercion is done as if sum
were performed

■ Collating sequence – a partial ordering

- $A < B < \dots < Y < Z$
- $0 < 1 < \dots < 8 < 9$
- either $\text{blank} < A, Z < 0$ or $\text{blank} < 0, 9 < A$

■ if lower-case letters exist:

- $a < b < \dots < y < z$
- either $\text{blank} < a, z < 0$ or $\text{blank} < 0, 9 < a$

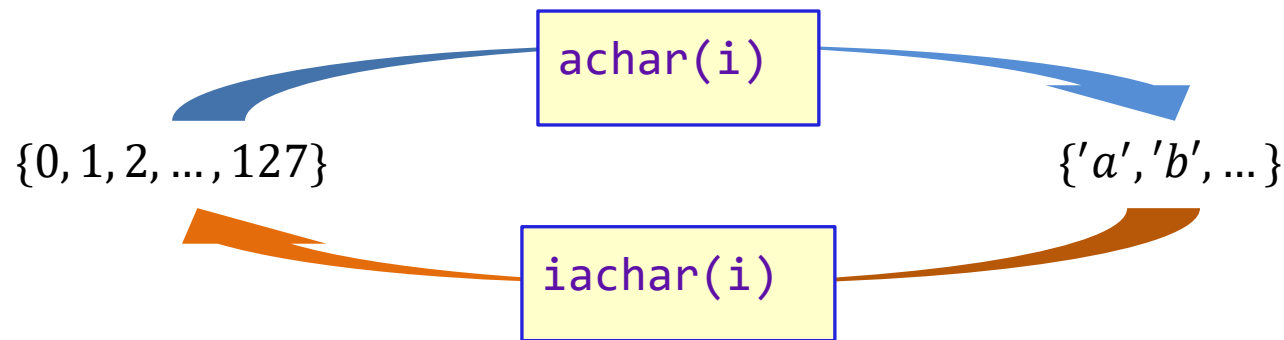


various definitions are possible (e.g., **ASCII**, EBCDIC) → do not rely on a particular ordering

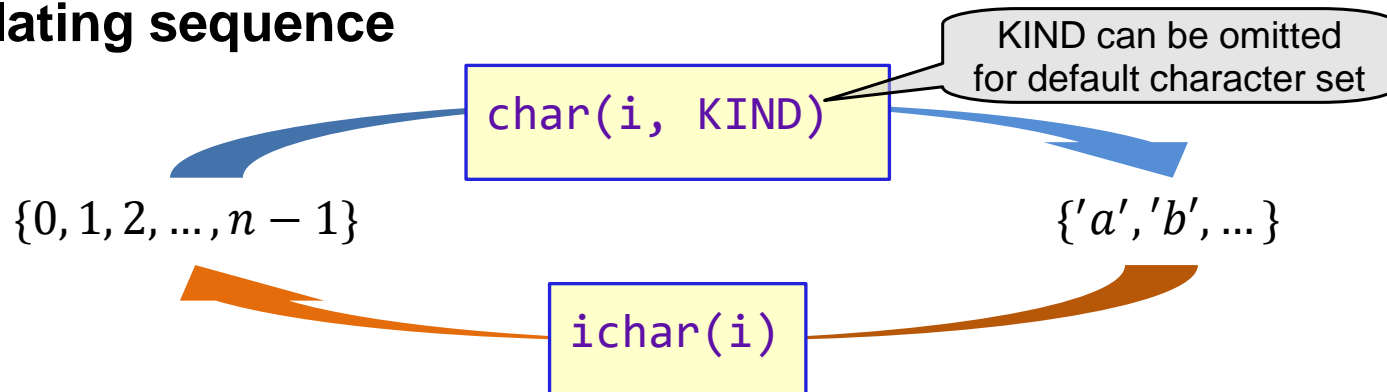
■ Character operands in relational expressions:

- must be of same kind
- strings are compared from left until a difference is found
- differing lengths: pad shorter string with blanks

- **Must use suitable intrinsics**
 - these operate on a single character
- **Mapping based on ASCII collating sequence**



- **Mapping for each character KIND based on the processor's collating sequence**



■ Intrinsic functions that operate on strings:

- default character kind
- comparison based on **ASCII collating sequence**
- return default logical result

■ Strings of different length:

- shorter one is padded on the right with blanks

```
lge(string_a,  
     string_b)
```

true if a follows b in the collating sequence or is equal, false otherwise

```
lgt(string_a,  
     string_b)
```

true if a follows b in the collating sequence, false otherwise

```
lle(string_a,  
     string_b)
```

true if b follows a in the collating sequence or is equal, false otherwise

```
llt(string_a,  
     string_b)
```

true if b follows a in the collating sequence, false otherwise

■ Note:

- zero-sized strings are identical

■ Only intrinsic operation:

- concatenation via //:

```
'AB' // 'cd'
```

- has the value 'ABcd'
- both operands of same kind
- length of the result is the sum of the length of the operands
- // can be iterated, is associative

■ Assignment of result

- to another character entity

■ Examples:

```
character(len=5) :: arg1, arg2
```

```
character(len=7) :: res1
```

```
character(len=12) :: res2
```

```
arg1 = 'no   '
```

```
arg2 = 'house'
```

substring of arg1

```
res1 = arg1(1:3) // arg2
```

```
! value of res1 is 'no hous'
```

```
res2(1:9) = arg1(1:3) // arg2
```

```
! value of res2(1:9)
```

```
! is 'no house '
```

note blank at end

- `res2` as a whole is **undefined** because `res2(10:12)` is undefined.

Now we proceed to the
exercise session ...



Subprogram units

■ Up to now,

- we've only written program units (main programs)

■ Disadvantages:

- replication of code (maybe even multiple times in the same program)
- difficult to navigate, edit, compile, test (maintainability issues)

■ Solution:

- functional encapsulation into **subprograms** (sometimes also called **procedures**)

■ Simple example:

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

dummy arguments

(declarations below):
only visible **inside** procedure

```
implicit none  
real :: a, b, c, x1, x2  
integer :: n  
: ! local variable declarations  
: ! calculate solutions  
end subroutine
```

- implementation calculates n , x_1 , x_2 from a , b , c

■ Three organization variants are possible

strongly
recommended

1. Put subprogram into a module program unit

- this is a container for the code
- the subprogram is then known as a **module procedure**



2. Implement it as an **internal subprogram**

- use a non-module program unit as container

3. Implement it as a „stand-alone“ **external subprogram**

- legacy coding style → risky to use, not recommended
- some discussion of this follows later, because you might need to deal with existing libraries



Module procedure

hosts the procedure

specifications come before **contains**

```
module mod_solvers
  implicit none
```

contains

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

```
  real :: a, b, c, x1, x2
```

```
  integer :: n
```

```
  : ! local variable declarations
```

```
  : ! calculate solutions
```

```
end subroutine
```

```
:
```

```
end module mod_solvers
```

... from previous slide

implicit none
is taken from
module specification part

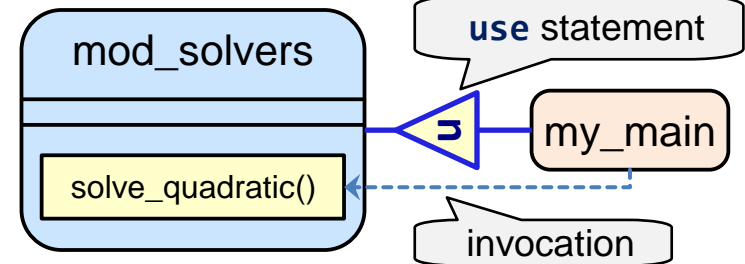
further module procedures
(solve_cubic, ...) may follow

- many more details on the semantics supported by Fortran modules will be incrementally provided

Invoking a module procedure (1)

■ From some **other** program unit

- **outside** the module – here a main program



```
program my_main
  use mod_solvers
  implicit none
  : ! declarations
  a1 = 2.0; a2 = 7.4; a3 = 0.2
  call solve_quadratic(a1, a2, a3, nsol, x, y)
  write(*, *) nsol, x, y
end program
```

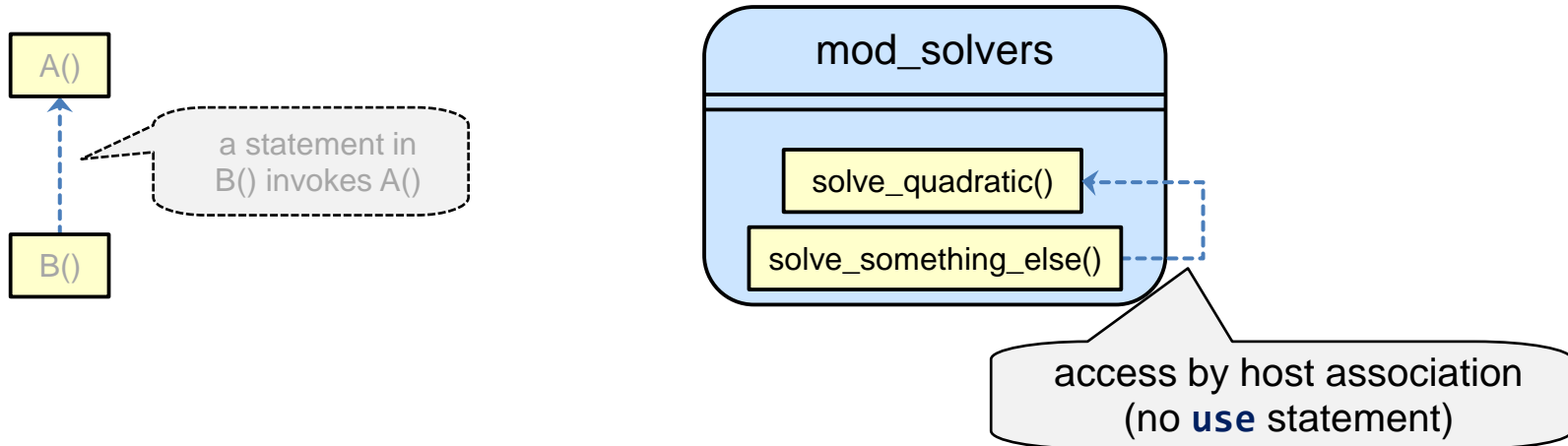
access **solve_quadratic** from module **mod_solvers** („**use association**“)

actual argument list

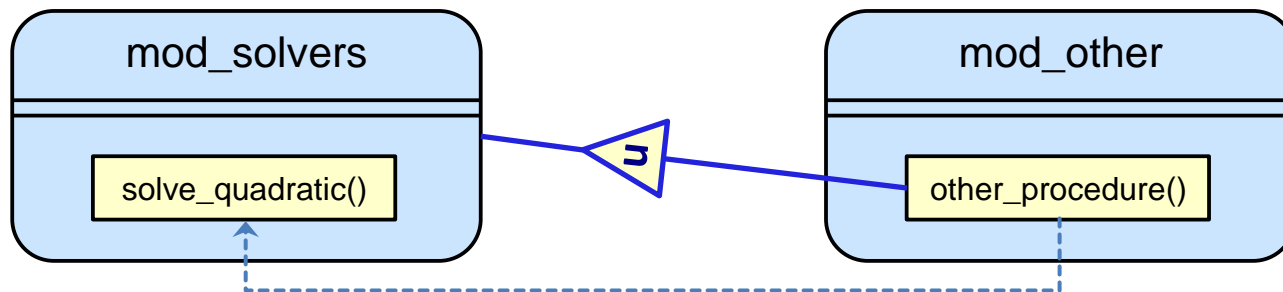
- the actual arguments **nsol** and possibly **x, y** are overwritten on each invocation

Invoking a module procedure (2)

- From some other module procedure in the same module



- From a module procedure in another module





■ Separate compilation

- different program units are usually stored in **separate** source files

■ Example: quadratic main program which calls procedure

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

must compile my_main **after** mod_solvers

```
gfortran -c -o my_main.o my_main.f90
```

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

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

link objects into executable

- **-c** specifies that **no linkage** should be performed; then, **-o** provides the **object file** name (default: same as source file name with extension replaced by **.o**),
- otherwise, **-o** specifies the **executable file** name.

■ Automated build systems for mass production:

- Example: (GNU) Make

- ... are automatically created for

1. module procedures and
2. internal procedures (discussed later),

- permit the compiler to do checking of **procedure characteristics** for each procedure invocation.

- This consists of checking the

1. type
2. kind
3. rank and other properties (discussed later)

of dummy arguments against those of actual arguments.



„stand-alone“ procedures have an **implicit interface**.
→ checking is not possible
→ some language features will not work at all

This is the reason for the compilation order mentioned previously.

- Mismatches cause rejection at **compile time**

■ Argument association

- each dummy argument becomes associated with its corresponding actual argument
- two variants:


1. Positional correspondence

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

for the above example: $a \leftrightarrow a1$, $b \leftrightarrow a2$, $x2 \leftrightarrow y$ 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

■ Establish (unsaved) local variables

- usually on the stack

- **Start with first executable statements of the subprogram**
 - and then continue execution from there;
 - this will usually reference and/or define each dummy argument.
 - The effect of argument association implies (essentially) that this behaves as if the corresponding actual argument were referenced and/or defined.

- **At the end of the subprogram, or when a RETURN statement is encountered**
 - delete local variables
 - remove argument association
 - for a subroutine: continue with first executable statement after the `call` statement

Note: dummy arguments are visible **only within the scope** of their defining procedure, and possibly within an enclosed scoping unit

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

■ Compile-time rejection of invalid code

- subroutine implementation:

```
real, intent(in) :: a
:  
a = ... ! rejected by compiler
```

- subroutine usage:

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

rejected by compiler

■ 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

■ 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)
    function result declaration
    real, intent(in) :: x, p
    : calculate function value and
      then assign to result variable
    wsqrt = ...
  end function wsqrt
end module
```

■ To be used in expressions:

```
use mod_functions
implicit none
real :: x1, x2, p, y
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

- **Alternative way of 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**

Optional arguments

Scenario:

- not all arguments needed at any given invocation
- reasonable default values exist

Example:

```
real function wsqrt(x, p)
  real, intent(in) :: x
  real, intent(in), optional :: p
  real :: p_loc
  if ( present(p) ) then
    p_loc = p
  else
    p_loc = 1.0
  end if
  :
end function wsqrt
```

path 1

path 2

- use of intrinsic logical function `present` is obligatory

Invocations:

```
y = wsqrt(x1, pg)
z = wsqrt(x2)
```

uses path 1

uses path 2

- in the second invocation, referencing dummy `p` (except via `present`) is non-conforming

Notes:

- optional arguments are permitted for functions and subroutines,
- an explicit interface is required,
- keyword calls are typically needed if a non-last argument is optional.

■ A procedure that invokes itself

- directly or indirectly
(may be a function or subroutine)

requires the **RECURSIVE** attribute

■ Example:

- Fibonacci numbers

result clause is
necessary here

```
recursive function fib(i) result(f)
  integer, intent(in) :: i
  integer :: f
  if (i < 3) then
    f = 1
  else
    f = fib(i-1) + fib(i-2)
  end if
end function fib
```

- this example demonstrates **direct** recursion

■ Note:

- since **F18**, the **recursive** attribute is not obligatory any more

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

internal function

invocation within host

a, b accessed from the host
→ host association

internal subroutine

slin is accessed by host association

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



Legacy functionality: statement function

```
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

Array dummy arguments – simplest case

Assumed size

often used in legacy libraries,
or calls to C

```
subroutine sscal ( N, SA, SX, INCX )
  integer, intent(in) :: N, INCX
  real, intent(in) :: SA
  real, intent(inout), dimension(*) :: SX
  :
end subroutine
```

executable statements

BLAS routine SSCAL $sx \leftarrow sa * sx$

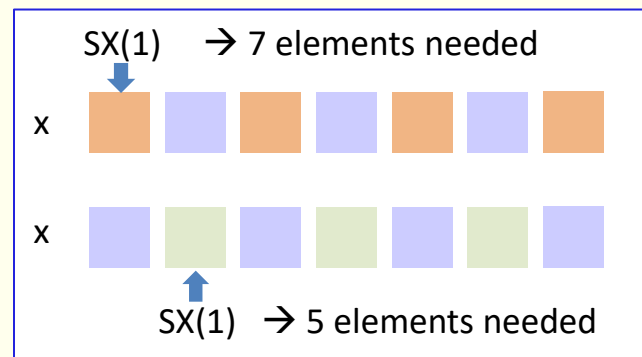
- N elements processed
- INCX: stride between subsequent elements

- SX: a contiguous storage sequence (here: up to $N * INCX$ elements needed)
- size of actual argument is assumed and must be sufficient to perform all accesses

Example invocations:

```
real :: x(7)
:
call sscal(4, 2.0, x, 2)
! overwrites "orange" elements
call sscal(3, -2.0, x(2), 2)
! overwrites "green" elements
```

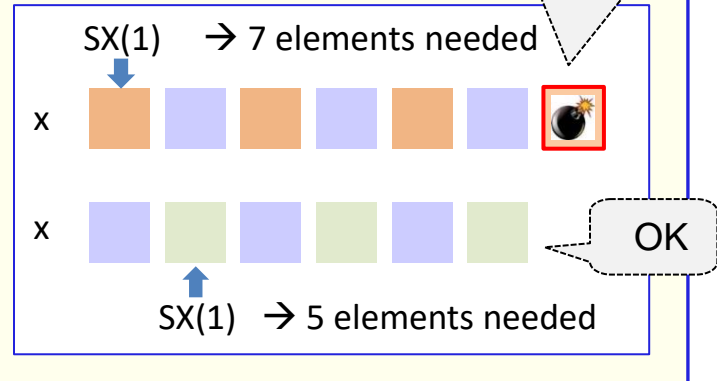
actual argument
provides „start address“



... about the size of the actual argument

illegal access attempted

```
real :: x(6)
:
call sscal(4, 2.0, x, 2)
! overwrites "orange" elements
call sscal(3, -2.0, x(2), 2)
! overwrites "green" elements
```



Possible consequences:

- program crashes immediately, or somewhat later, or
- element of another array is overwritten → incorrect result, or
- you're lucky, and nothing bad happens (until you start using a different compiler, or other compiler options)

An improved way of passing arrays will be shown tomorrow

■ 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
```



Side effects in procedure calls

■ Procedure definition

```
subroutine modify_a_b(a, b)
  real, intent(inout) :: a, b

  ...
  a = ...
  ...
  b = ...
end subroutine
```

■ ... and invocation

```
real :: x, y
...
x = ...
y = ...

call modify_a_b(x, y)

call modify_a_b(x, x)
```



■ Second call:

- **aliases** its dummy arguments
- how can two results be written to a single variable? (same memory location!)

Definition of aliasing

■ Aliasing of dummy argument:

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

concept is explained later

■ Example for 3.:

```

program alias_host
  real :: x(5)

  call bar(x,5)
contains
  subroutine bar(this,n)
    real :: this(*)
    integer :: n

    ...
  end subroutine bar
end program

```

x is accessible by host association

- inside bar(), this is aliased against x

■ Procedure definition

```
subroutine modify_a(a, b)
  real, intent(inout) :: a
  real, intent(in) :: b
```

A a = 2 * b

B ... = b

C a = a + b

```
end subroutine
```

■ ... and invocation

```
real :: x, y
...
x = ...
y = ...
call modify_a(x, y)
call modify_a(x, x)
call modify_a(x, (x))
```



■ Second call: aliased

- next slide discusses what might happen (potential conflicts of reads and writes)



Implementation dependence

- on argument passing mechanism
- assume $x=2.0$, $y=2.0$ at entry

Model 1: copy-in/copy-out

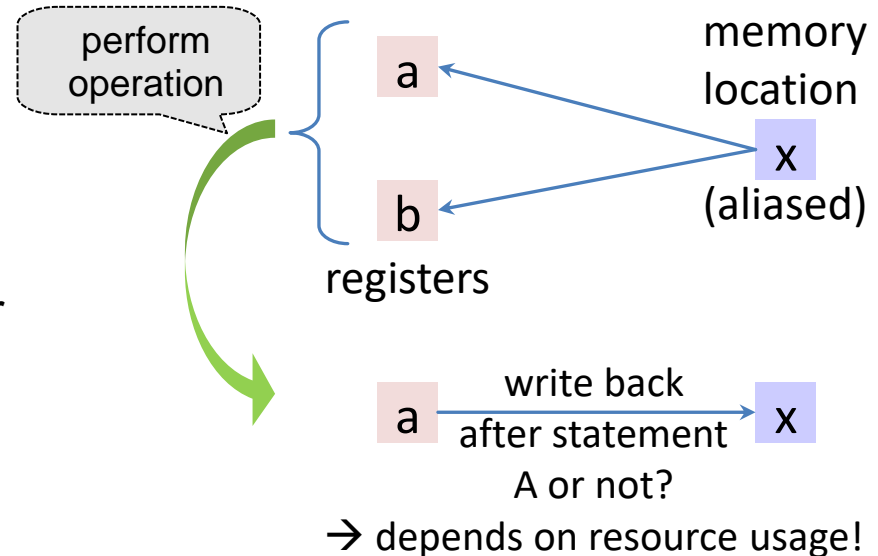
- working on local copies
- both aliased and non-aliased calls produce the same result for x (6.0)
- only first argument is copied out
- third call always effectively uses copy-in for the second argument (actual argument is an expression) → avoids aliasing

Model 2: call-by-reference

- pass address of memory location

other models are conceivable

- result depends on procedure-internal optimization



- possible results: 6.0 or 8.0
- further possible side effect: result of statement B depends on statement reordering

■ **Consequence:**

- restriction in language which makes the problematic calls illegal
- but aliasing is not generally disallowed

■ **Restriction:**

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

■ **Intent:**



enable **performance optimizations** by statement reordering and/or register use

- avoid ambiguities in assignments to dummy arguments

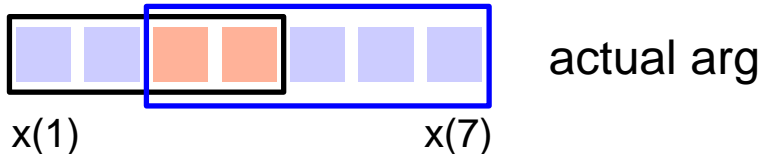
■ **Notes:**

- further rules exist that apply to dynamic features of the language
→ see advanced course
- exceptions to restrictions exist for special situations
→ see advanced course
- restriction effectively also applies to parallel procedure invocations in a shared memory environment (e.g., OpenMP)

Aliasing – further examples (rather artificial)

Partial aliasing:

```
real :: x(7)
x = ...
call subp(x(1:4), x(3:))
```



- `x(3)`, `x(4)` may not be modified by `subp()` via either dummy argument
- `x(1:2)` may be modified via the first argument
- `x(5:7)` may be modified via the second argument

(assuming that `subp()` always references complete argument)

Aliasing against host associated entity:

```
program alias_host
  real :: x(5)

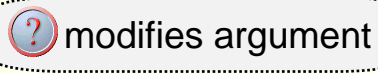
  call bar(x,5)
contains
  subroutine bar(this,n)
    real :: this(*)
    integer :: n
    this(1) = ...
    ... = x(1) ! NO
    ... = x(2) ! OK
  end subroutine bar
end program
```

- `this(2:5)` is not modified by `bar()`





Example function

```
integer function badfun(i)
  integer, intent(inout) :: i
  :
  i = -1
  badfun = ...
end function
```

 modifies argument


Undefined actual argument

```
if ( x < 0.0 .or.      &
     badfun(i) > 0 ) then
  ...
end if
```


  i is **undefined** here

- because `badfun()` may or may not have been called

Effective aliasing:



```
if (i < 0 .and. &
    badfun(i) > 0) then
  ...
end if
```



```
q = badfun(i) + badfun(i)**2
```

- **Restriction:** a function reference is not allowed to modify a variable or affect another function reference appearing in the same statement

→ above invocations are non-conforming

■ Strategy 1:

- **document** proper usage
- for the previous example, an invocation like

```
q = badfun( i ) + badfun( j )**2
```

with separate actual arguments would be OK.

■ Strategy 2 (preferred):

- avoid side effects altogether
- at minimum, declare all dummy arguments of a function INTENT(IN).
- even better: declare all functions PURE (see next slide)

Example:

```
pure integer function goodfun(i)
  integer, intent(in) :: i
  :
  goodfun = ...
end function
```

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
- ...

troublesome for debugging
→ temporarily remove the attribute

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

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

■ For subroutines declared PURE, the only difference from functions is:

- all dummy arguments must have declared INTENT

■ Notes on PURE procedures in general:

- Purposeful use of the PURE property in an invocation requires 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.

■ Use 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

■ 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



Interface specifications and Procedures as arguments

Recall BLAS example (SSCAL)

- **BLAS** is a „legacy library“, but very often used
 - „stand-alone“ **external** procedures with implicit interfaces
 - baseline (seen often in practice): unsafe usage – no signature checking

```
program uses_sscal
  implicit none
  external :: sscal
  real :: x(7)
  call sscal(4, 2.0, x, 2)
  call sscal(3, -2, x(2), 2)
  write(*,*) x
end program
```

statement often omitted
→ **sscal** external by default



no complaint from compiler
about wrong type of actual argument

- another common error: argument count wrong

■ Note:

- for external functions, the return type must be explicitly declared if strong typing is in force.

Manually created explicit interface

(remember: this is neither needed nor permitted for module procedures!)

- Makes external procedures safer to use

- Recommendation:

- place in **specification part** of a module

```
module blas_interfaces
  interface
    subroutine sscal ( N, &
                      SA, SX, INCX )
      integer, intent(in) :: N, INCX
      real, intent(in) :: SA
      real, intent(inout), &
        dimension(*) :: SX
    end subroutine
    : ! further
    : ! interfaces
  end interface
end module
```

Interface block

executable statements
are **not permitted**

- Modified program that invokes the procedure

```
program uses_sscal
  use blas_interfaces
  implicit none
  real :: x(7)
  call sscal(4, 2.0, x, 2)
  call sscal(3, -2, x(2), 2)
  write(*,*) x
end program
```

rejected by compiler

- similarly, incorrect argument count is now caught by the compiler
- however, incorrect array size is usually not

■ Additional language feature needed:

- interoperability with C; intrinsic module ISO_C_BINDING

■ Example: C function with prototype

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

■ Fortran interface:

```
module libm_interfaces
  implicit none
  interface
    real(c_float) function lgammaf_r(x, is) BIND(C)
      use, intrinsic :: iso_c_binding
    end function
  end interface
end module
```

enforce C name mangling

provides kind numbers for interoperating types

C-style value argument

■ KIND numbers:

- `c_float` and `c_int` are usually the default Fortran KINDs anyway
- further types supported: `c_char` (length 1 only), `c_double`, ...
- unsigned types are **not** supported

■ Mixed-case C functions

- an additional label is needed

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

```
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
```

invocation from Fortran via Fortran name

■ C-style arrays

- require assumed size declaration in Fortran interface

■ Much more information is provided in the advanced course

Procedures as arguments (1)

Up to now:

- procedure argument a variable or expression of some datatype

For a problem like, say, numerical integration

input data:

- interval [a, b]
- function f(.)



output:

$$\int_a^b f(t) dt$$

- want to be able to provide a complete function as argument
- „functional programming style“

Example:

- implementation of quadrature routine

```

module quadrature
  implicit none
  contains
  subroutine integral_1d( &
    a, b, fun, valint, status )
    real, intent(in) :: a, b
    real, intent(out) :: valint
    integer, optional, &
      intent(out) :: status
  interface
    real function fun(x)
      real :: x
    end function
  end interface
  : ! implementation
  ... = ... + fun(xi) * wi
  valint = ...
end subroutine
end module

```

invokes function that is provided as actual argument

■ Invoking the quadrature routine

- step 1 – provide implementation of integrand

```
module integrands
  implicit none
contains
  real function my_int(x)
    real :: x
    my_int = x**3 * exp(-x)
  end function
end module
```

- step 2 – call quadrature routine with suitable arguments

```
program run_my_integration
  use integrands
  use quadrature
  implicit none
  real :: a, b, result

  a = 0.0; b = 12.5
  call integral_1d(a, b, &
                  my_int, result)
  write(*, *) 'Result: ', &
             result
end program
```

■ Dummy procedure interface

- writing this may be cumbersome if specification must be reiterated in many calls
- note that no procedure needs to actually exist as long as no invocation has been written → interface is „abstract“

■ Equivalent alternative

- define the abstract interface in specification part of the module and reference that interface (possibly very often)

Now we proceed to an **exercise session ...**

```
module quadrature
  implicit none
  abstract interface
    real function f_simple(x)
      real :: x
    end function
  end interface
contains
  subroutine integral_1d( &
    a, b, fun, valint, status )
    real :: a, b, valint
    integer, optional :: status
    procedure(f_simple) :: fun
    ! implementation
    ... = ... + fun(xi) * wi
    valint = ...
  end subroutine
end module
```

alternative implementation

reference to above definition



Derived Types and more on Modules

Overcome insufficiency

- of intrinsic types for description of abstract concepts

```
module mod_body
  implicit none
  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

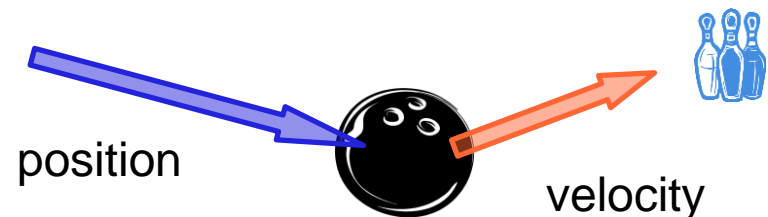
Type components:

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

Recommendation:

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

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



layered creation of more complex types from simple ones

■ Objects of derived type

■ Examples:

```
use mod_body
type(body) :: ball, copy
type(body) :: asteroids(ndim)
```

here: a program unit
outside `mod_body`

- 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!)

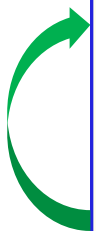
■ Default assignment

```
copy = ball
```

- copies over each type component individually

■ Implementation of „methods“

```
module mod_body
  implicit none
  type :: body
  ...
contains
  subroutine kick(this, ...)
    type(body), intent(inout) :: this
    ...
  end subroutine
end module
```



type definition
shown earlier

```
use mod_body
type(body) :: ball
type(body) :: asteroids(ndim)
... ! define objects
call kick(ball, ...)
call kick(asteroids(j), ...)
```

- declares scalar dummy argument of `type(body)`
- access to type definition here is by host association
- invocation requires an actual argument of exactly that type (→ explicit interface required)

■ Via component **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)

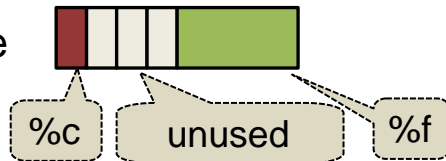
Remarks on storage layout

Single derived type object

- compiler might insert padding between type components

```
type :: d_type
  character :: c
  real :: f
end type
```

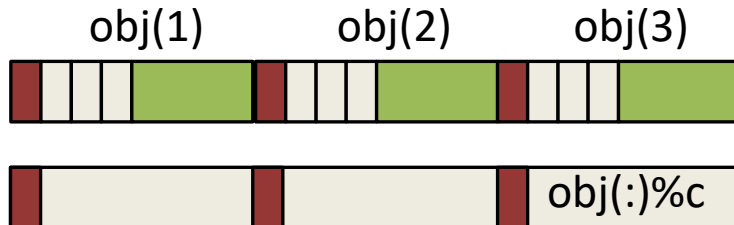
could look like



Array element sequence

- as for arrays of intrinsic type

```
type(d_type) :: obj(3)
```



Special cases

avoid use



sequence types **enforce** storage layout in specified order

```
type :: s_type
  sequence
  real :: f
  integer :: il(2)
end type
```

- BIND(C)** types **enforce** C struct storage layout:

```
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;
```

Semantics

- **Permits packaging of**
 - **global variables**
 - named constants
 - type definitions
 - procedure interfaces
 - procedure implementations

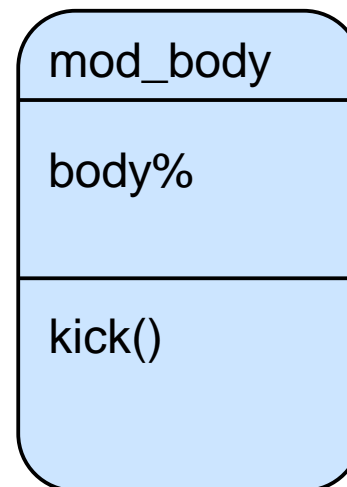
for reuse,

- **Allows**
 - **information hiding**
 - (limited) **namespace management**

Module definition syntax

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

Symbolic representation



reference:
example
from
earlier
slide

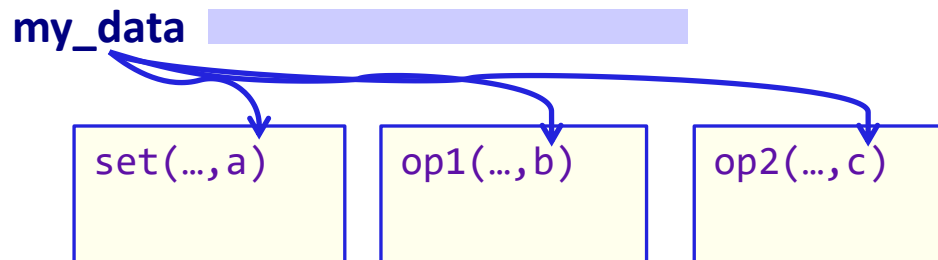
An alternative for communicating with subprograms

■ Typical scenario:

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

■ Known mechanism:

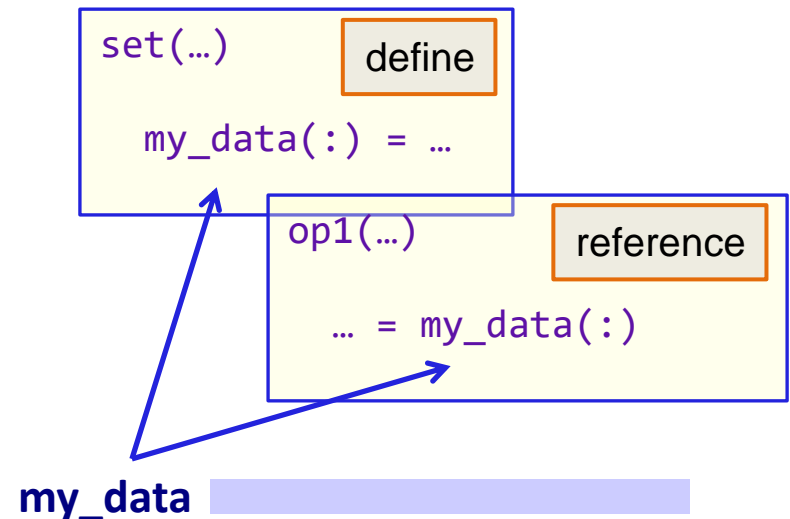
- data are passed in/out as procedure arguments



- **disadvantage:** need to declare in exactly one calling program unit; access not needed from any other program unit (including the calling one)

■ Alternative:

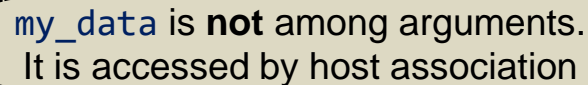
- define **global storage area** for data
- accessible from subroutines without need for the client to provision/manage it



- better separation of concerns

Declaring and using a global variable

```
module mod_globaldata
  implicit none
  integer, parameter :: dm = 10000
  real :: my_data(dm)
contains
  subroutine set(...)
    ...
    my_data(:) = ...
  end subroutine set
  subroutine op1(...)
    ...
  end subroutine op1
end module mod_globaldata
```



my_data is **not** among arguments.
It is accessed by host association

- **Assumption: data in question only need to exist once**
 - sometimes also called „Singleton“ in computer science literature
- **Further attributes can be specified** (discussed later)
- ⚠ **Fortran 77 COMMON blocks should **not** be used any more**

F18 declares COMMON obsolescent

Information hiding (1)

■ Prevent access to `my_data` by use association:

```

module mod_globaldata
  implicit none
  integer, parameter :: dm = 10000
  real, private :: my_data(dm)
contains
  ...
end module mod_globaldata

```

procedures `set`, `op1`, ... as in previous slide

- refers to access **by name**
- default accessibility is **public**

```

use mod_globaldata
my_data(5) = ...
call set(...)

```

`my_data` is private → **rejected** by compiler

`set()` is public → OK

Information hiding (2)

■ Changing the default accessibility to private

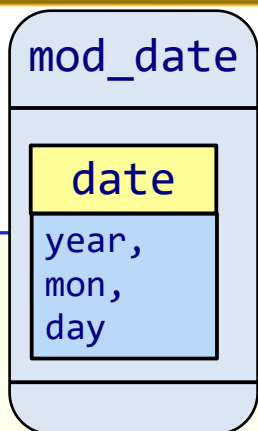
```
module mod_globaldata
  implicit none
  private
  public :: set, op1, ...
  integer, parameter :: dm = 10000
  real :: my_data(dm)
contains
  ...
end module mod_globaldata
```

blanket private statement

- need to explicitly declare entities **public** that should be accessible by use association

Hide components

```
module mod_date
  implicit none
  type, public :: date
    private
    integer :: year, mon, day
  end type
contains
...
end module mod_date
```



- type is **public**, but its components are **private** → access to type components or use of structure constructor requires access by host association
- default assignment is permitted in use association context

Write a module function

```
module mod_date
...
contains
  function set_date(year, &
    mon, day) result(d)
    type(date) :: d
    ...
    d = date(year, mon, day)
  end function
end module mod_date
```

Usage example:

```
use mod_date
type(date) :: easter
components private – rejected by compiler
easter = date(2016,03,27)
public function set_date – OK
easter = set_date(2016,03,27)
```

■ Some type components PRIVATE, others PUBLIC

F03

```
module mod_person
  use mod_date
  ...
  type, public :: person
    private
    character(len=smx) :: name
    type(date) :: birthday
    character(len=smx), public :: location
  end type
  ...
end module
```

● Usage example:

```
use mod_person
type(person) :: a_person
a_person%name = 'Matthew'

a_person%location = 'Room 23'
```

name is private – **rejected** by compiler

location is public – **OK**

■ „Read-only“ flag that can be applied to module variables

```
module mod_scaling
  implicit none
  real, protected :: conversion_factor = 11.2
contains
  subroutine rescale(factor)
    ...
    conversion_factor = conversion_factor * factor
  end subroutine
end module
```

modification OK because in host

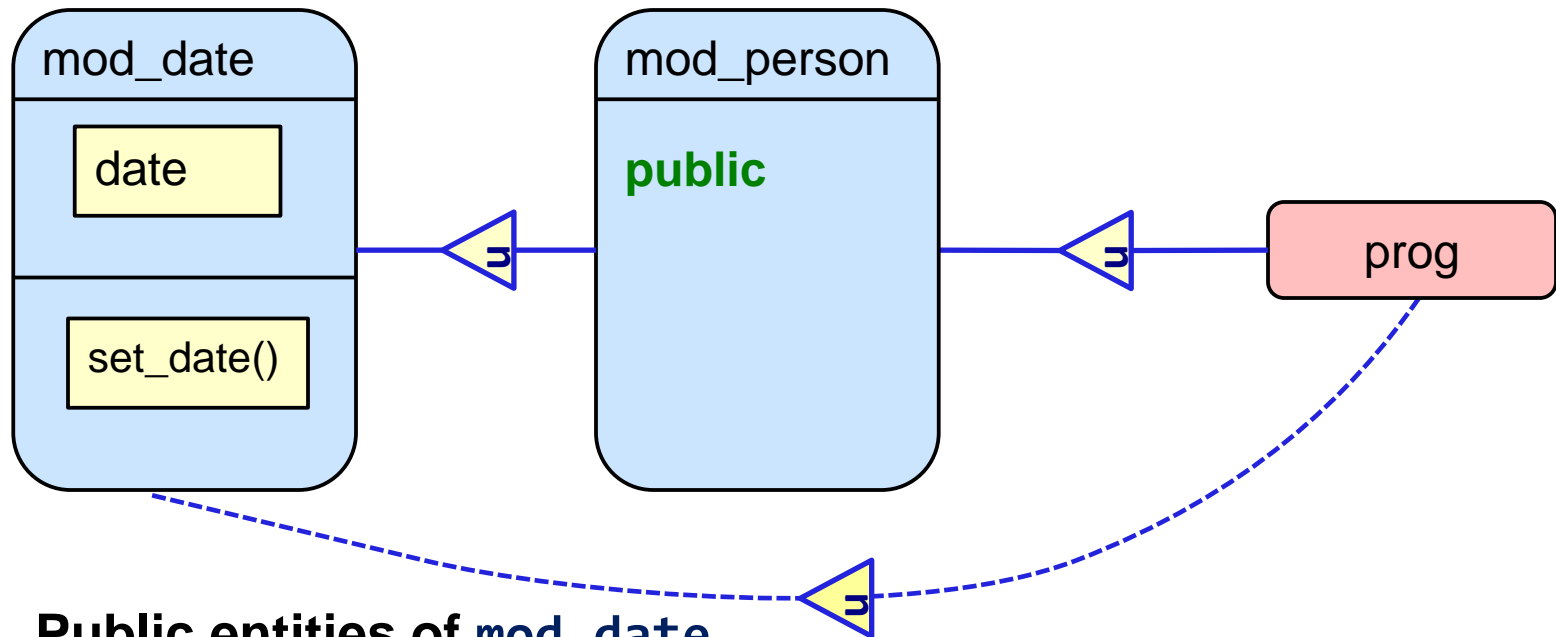
- modification of variable value only permitted in host association context

```
use mod_scaling
...
conversion_factor = 3.5
call rescale(1.1)
x_new = x_old * conversion_factor
```

non-conforming – likely **rejected** by compiler

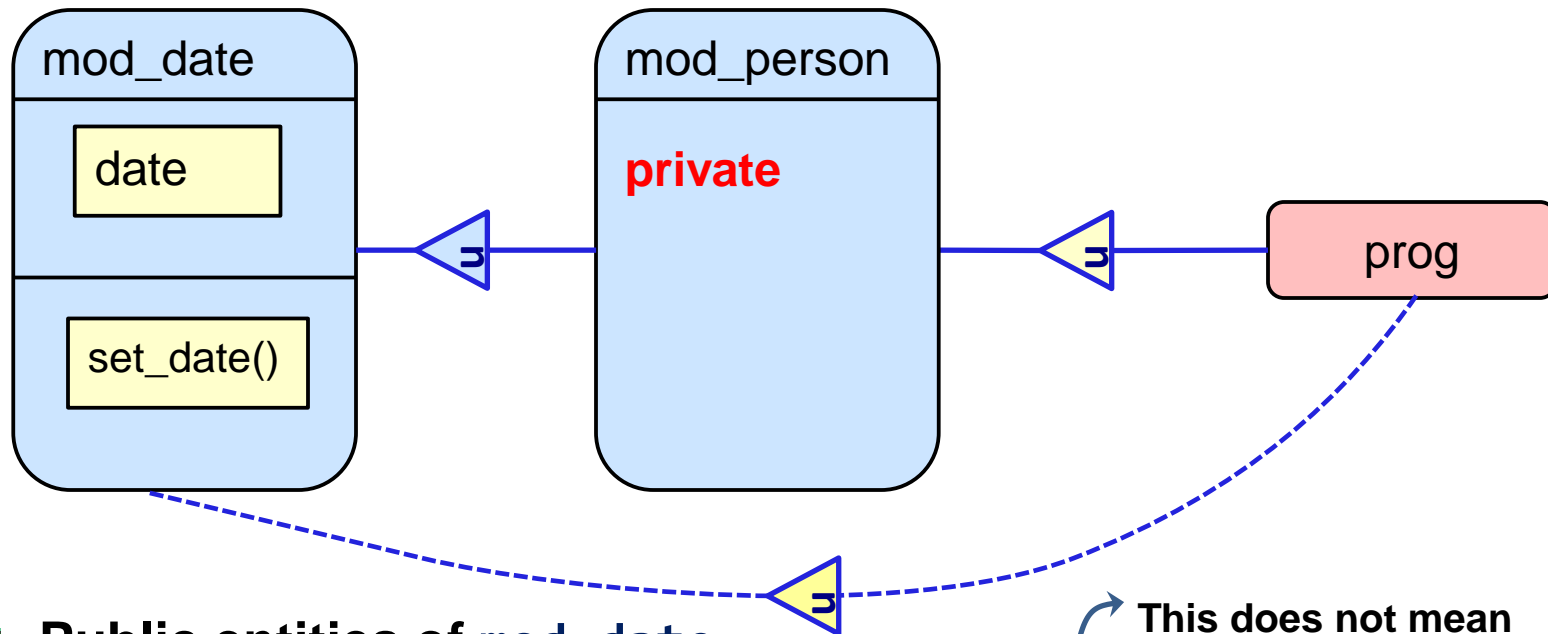
OK

read access is permitted



■ Public entities of `mod_date`

- can be accessed inside host of `mod_person`
- can also be accessed inside host of `prog` due to the blanket `public` statement
- **Note:** access can be changed from `public` to `private` for individual entities from `mod_date` inside `mod_person`. But this will have no effect if the associating unit directly uses `mod_date` (dotted line)



Public entities of mod_date

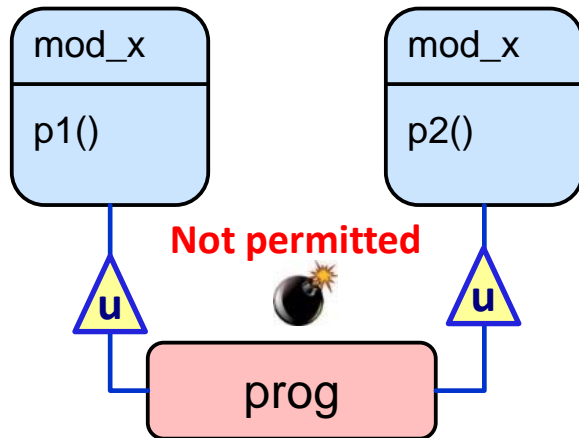
- can be accessed inside host of mod_person
- **cannot** be accessed inside host of prog due to the blanket private statement
- **Note:** access can be set to public for individual entities from mod_date inside mod_person

This does not mean

- that date and set_date() are private per se,
- since prog may still access them by using mod_date directly (dotted line)

Global identifiers

- for example, module names
- must be **unique** for program



Local identifiers

- for example, names declared as variables or type names or procedure names („class 1“)
- must be unique for scoping unit

```
program prog
  use mod_date
  implicit none
  integer :: date(3)
  ...
end program
```

has type definition of date

collision between use associated type name and variable name → non-conforming

Exception:

- generic procedure names
- discussed tomorrow

How to avoid name space issues for local identifiers



1. Use information hiding to encapsulate entities only needed in host
 - i.e. the PRIVATE attribute
2. Adopt a **naming convention** for public module entities
3. **Rename** module entities on the client
4. **Limit access** to module entities on the client
5. Limit the number of scoping units that access a module

Some or all of the above can be used in conjunction

Some possible naming conventions

Scheme 1

- **Module name**
 - `mod_<purpose>`
- **Data type in module**
 - `<purpose>`
 - `<purpose>_<detail>` if multiple types are needed
- **Public variables / constants**
 - `var_<purpose>_<detail>`
 - `const_<purpose>_<detail>`
- **Public procedures**
 - `<verb>_<purpose>` or
 - `<verb>_<purpose>_<detail>`

Example: module `mod_date`

Scheme 2

- **Module name**
 - `<name>`
- **Data type in module**
 - `<name>_<purpose>`
- **Public variables / constants**
 - `<name>_<purpose>`
- **Public procedures**
 - `<name>_<verb>` or
 - `<name>_<verb>_<purpose>`

Example: modules `mpi`, `mpi_f08`

■ Corrected example from previous slide

```
program prog
  use mod_date, pdate => date
  implicit none
  type(pdate) :: easter
  integer :: date(3)
  ...
end program
```

type has been renamed,
but works with all semantics
defined in `mod_date`

■ Avoiding naming collisions that result from use association only

```
program prog
  use mod_date
  use otherdate, pdate => date
  implicit none
  type(date) :: easter
  type(pdate) :: schedule
  ...
end program
```

also has type
definition of `date`

collision is triggered only if entity is
actually referenced on the client

■ Assumption:

- mod_date contains a public entity lk

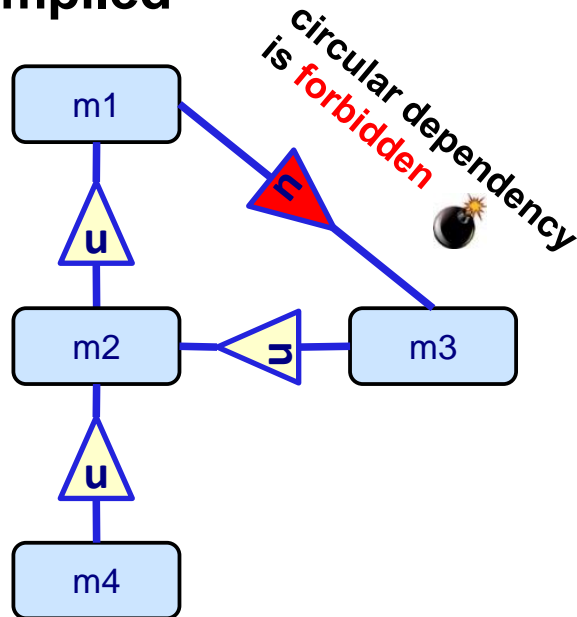
```
program prog
  use mod_date, only : date
  implicit none
  integer, parameter :: lk = ...
  type(date) :: easter
  ...
end program
```

■ Combine ONLY with renaming

```
program prog
  use mod_date, only : &
                        pdate => date
  implicit none
  integer, parameter :: lk = ...
  type(pdate) :: easter
  real :: date(3)
  ...
end program
```

- avoid collision via ONLY option that limits use access to specified entities
- works if none of the needed entities has a collision

■ Modules are separately compiled



■ If a program unit use associates a module

- the latter must be compiled first
- directed acyclical dependency graph („DAG“)

- order of compilation in the above setup:
- m1, m2, [m3|m4]
- dependency generation support for build systems is useful

■ Circular use dependencies are disallowed

- example: m1 may not use m3, since m3 (indirectly) uses m1

■ Recompilation cascade:

- if a module is changed, all program units using it **must** be recompiled
- usually even if only the implementation (`contains` part) is modified

solution to this in **advanced** course

■ At compilation

- the usual object file is generated
- per module contained in the file, one additional file with information describing at least the specification part of the module, including the signatures of all explicit interfaces, is created
- this **module information file** usually is named *module_name.mod*; it is essentially a kind of pre-compiled header
- it is needed whenever the compiler encounters a `use <module_name>` statement in another program unit → potentially **forces compilation order**

F08

Note: large modules and multitudes of dependent modules can cause problems
→ use submodules to deal with this (cf. advanced topics course)

■ Location of module information files

- need to use the compiler's `-I<path>` switch if not in current directory (usually the case for packaged libraries, but the files should be placed in the `include` folder instead of `lib`)



■ Assumption

- a (possibly large) group of object files covering a certain area of functionality was generated
- should be packaged up for later use (possibly by someone else)

■ Generate a library

- use the archiver `ar`

```
ar -cru libstuff.a a.o b.o c.o
ar -cru libstuff.a d.o
ranlib libstuff.a
```

- options: `-c` creates library archive if necessary, `-r` replaces existing members of same name, `-u` only does so if argument object is newer
- `ranlib` generates an archive index

■ Further notes

- objects from different (processor) architectures should not go into the same library file
- some architectures support multiple binary formats – especially 32 vs. 64 bit
 - special options for the `ar` command may be needed (for example AIX on Power: `-Xany`)
- shared libraries: not treated in this course



■ Assumption

- prepackaged library `libstuff.a` is located in some directory, say `/opt/pstuff/lib`

■ How to make use of objects inside library?

- task performed by the linker `ld`
- normally: **implicitly** called by the compiler

```
ifort -o myprog.exe myprog.o \  
      -L/opt/pstuff/lib -lstuff
```

- complex dependencies: **multiple** libraries may be required

■ What can go wrong?

- error message about **missing** symbols
→ need to specify additional libraries, or fix linkage order
- error message or warning about **duplicate** symbols
→ may need to fix linkage line e.g., by removing superfluous libraries
- error message concerning binary incompatibility (32-bit vs. 64-bit binaries)
→ need to specify libraries appropriate for used compilation mode



Array Processing

More on array declarations

Previously shown array declarations: Rank 1

- however, higher ranks (up to 15) are possible (scalars have rank 0)
- permit representation of matrices (rank 2), physical fields (rank 3, 4), etc.

Example: Rank 2 array

```
integer, parameter :: nb = 2, ld = 1
real, dimension(nb, -ld:ld) :: bb
```

dimensions must be **constants** for „static“ arrays

- lower bounds: 1, -1
- upper bounds: 2, 1
- shape**: 2, 3

if no lower bound is specified, it has the value 1

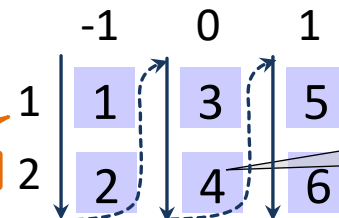
i-th element of the shape is also called i-th **extent**

- size**: $2 * 3 = 6$

- layout in memory:

„column major“ array element sequence

1st dimension



2nd dimension

bb(2,0) is fourth element in the sequence

■ Bounds

```
lbound(array [, dim])
```

```
ubound(array [, dim])
```

- lower and upper bounds
- without `dim`, a default rank 1 integer array with bounds in all dimensions is returned, else the bound in specified `dim`
- special cases will be mentioned as they come along ...

■ Shape and size

- rank 1 array with shape of array or scalar argument (for a scalar, a zero size array)

```
shape(source)
```

- size of array (or extent along dimension `dim` if present)

```
size(array [, dim])
```

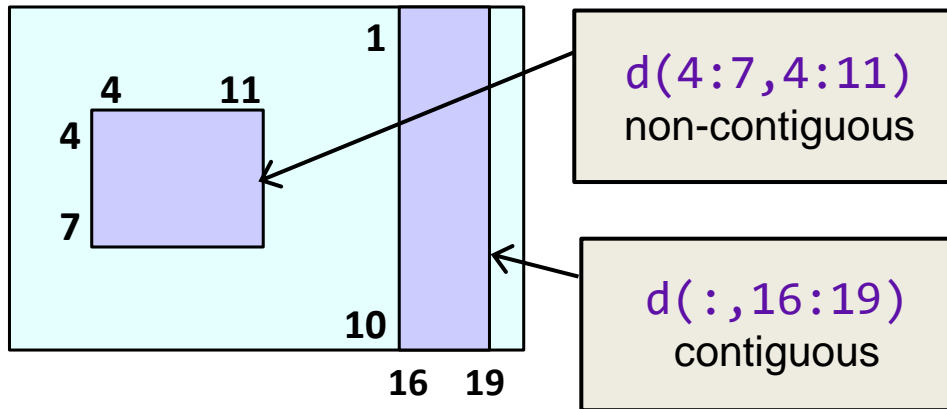
- Note:
 $\text{extent} = \text{ubound} - \text{lbound} + 1$

```
real :: d(10, 20)
```

Array sections (1)

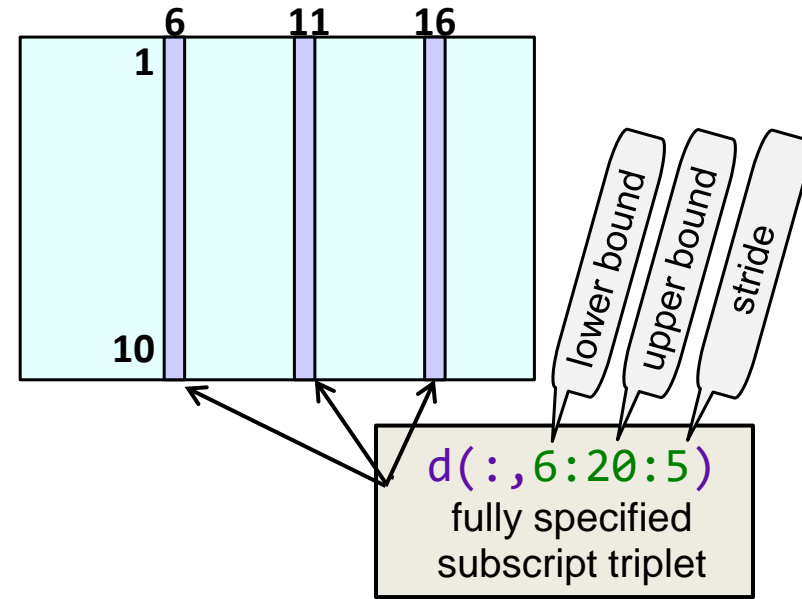
■ Array subobject

- created by **subscript** specification



- a colon without bounds specifications means the **complete** set of indices in the dimension it is specified in
- also possible: only lower or only upper bounds are specified in the subscript

■ Strided array subobject



- it is allowed to omit index specifications:

```
d(:, ::2)
```

every second column of `d`, starting in the first one

■ Array constructor

```
intrinsic :: reshape  
real :: a(6)  
a = [ 1.,3.,5.,7.,9.,11. ]  
bb = reshape([ 1,3,5,7,9,11 ], &  
             shape=[ 2,3 ])
```

alternative notation: (/ ... /)

- used for defining complete arrays (all array elements)
- intrinsic `reshape` creates a higher rank array from a rank 1 array

■ Array assignment

- conformability of LHS and RHS: if RHS is not a scalar, shape must be the same

```
bb = d(4:5,16:18)
```

we don't care about lower bounds here

```
d(:,11:) = d(:,5:14)
```

overlap of LHS and RHS
→ array temporary may be created

- scalars are broadcast
- element-wise assignment **by array element order**

Array sections (2): Vector subscripts

■ A rank 1 integer **expression** for subobject extraction

- one-to-one:

```
... = v( [ 2, 3, 9, 5 ] )
```



- many-to-one:

```
... = v( [ 2, 3, 9, 2 ] )
```



- you can also use an integer array variable as vector subscript:

```
iv = [ 2, 3, 9, 5 ]  
... = v( iv )
```

■ Care is needed in some cases:

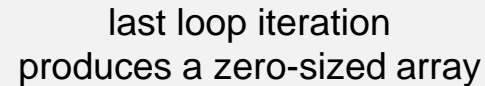
- `v(iv)` cannot appear in a context that may cause ambiguities e.g., as an actual argument matching an INTENT(INOUT) dummy

■ Zero-size arrays

- may result from suitable (algorithm-induced) indexing of a defined array, or by dynamic allocation (discussed later)
- always defined, but no valid reference of an array element is possible
- lower bound is 1, upper bound 0

■ Example:

```
do i = 1, n
  :
  ... = d(:,i:n-1)
end do
```



last loop iteration
produces a zero-sized array

- avoids the need for explicit masking
- remember array conformity rules

■ Subarray formation may change rank of object:

```
real :: e(10, 10, 5, 20) ! rank 4
```

number of array elements: 10000

```
... = e( [ 2, 3 ], 5, :, 20) ! rank 2
```

number of array elements: 10

vector subscript

subscript triplet (same as :::)

- number of vector subscripts and subscript triplets determines rank of subarray

```
real :: f(2, 5)
```

```
f = e( [ 2, 3 ], 5, :, 20)
```

this assignment is equivalent to:

```
f(1,1) = e(2, 5, 1, 20)
```

```
f(2,1) = e(3, 5, 1, 20)
```

```
f(1,2) = e(2, 5, 2, 20)
```

```
...
```

```
f(1,5) = e(2, 5, 5, 20)
```

```
f(2,5) = e(3, 5, 5, 20)
```

- **Note:** declaration syntax and that used in executable statements have different meanings!

■ Earlier declarations ...

```
type :: body
...
  real :: pos(3)
end type
type(body) :: asteroids(ndim)
```

- However, there may not be two (or more) designators which are arrays:

<code>asteroids(:)%pos</code>

disallowed

■ Subobject designators:

<code>asteroids(2)%pos(2)</code>	real scalar
<code>asteroids(2)%pos</code>	real rank-1 array
<code>asteroids(:)%pos(3)</code>	real rank-1 array
<code>asteroids(2)</code>	scalar of type body (with array subobjects)

Array expressions

Illustrated by operations on numerical type

- operations are performed **element-wise**
- binary operations of scalar and array: each array element is one operand, the scalar the other
- binary operations of two conformable arrays: matching array elements are the operands for result array element

Lower bounds of expressions

- are always remapped to 1!

example:
a(6,11) is assigned
the value b(1,1) * a(4,2)

```

real :: a(10, 20), b(5, 10)
intrinsic :: all, sqrt

b = b + 1.0 / a(1:5,1:10)

if ( all( a >= 0.0 ) ) then
  a = sqrt(a)
end if

a(6:10,11:20) = &
  b * a(4:8,2:11)

```

subarray of a conformable with b

a logical 10 by 20 array

elemental intrinsic

Array intrinsics that perform reductions

Name and arguments	Description	Name and arguments	Description
<code>all(mask [, dim])</code>	returns <code>.true.</code> if all elements of logical array mask are true, or if mask has zero size, and <code>.false.</code> else	<code>minval(array [, dim] [, mask])</code>	returns the minimum value of all elements of an integer or real array. For a zero-sized array, the largest possible magnitude positive value is returned.
<code>any(mask [, dim])</code>	returns <code>.true.</code> if any element of logical array mask is true, and <code>.false.</code> if no elements are true or if mask has zero size.	<code>product(array [, dim] [, mask])</code>	returns the product of all elements of an integer, real or complex array. For a zero-sized array, one is returned.
<code>count(mask [, dim])</code>	returns a default integer value that is the number of elements of logical array mask that are true.	<code>sum(array [, dim] [, mask])</code>	returns the sum of all elements of an integer, real or complex array. For a zero-sized array, zero is returned.
<code>maxval(array [, dim] [, mask])</code>	returns the maximum value of all elements of an integer or real array. For a zero-sized array, the largest possible magnitude negative value is returned.	<code>parity(mask [, dim])</code>	returns <code>.true.</code> if <code>.neqv.</code> of all elements of logical array mask is true, and <code>.false.</code> else.

F08

9 transformational functions

- except for `count`, result is of same type and kind as argument
- ninth function is on next slide ...

Additional optional arguments

- provide extra semantics
- see following slides

■ Increased abstraction:

- programmer can define operation to use
- it can be applied to objects of arbitrary type

■ Example invocation:

```
use mod_p
```

```
type(p) :: a(ndim), b
```

a procedure argument

```
b = reduce(a(i:j), prod_p, &  
          identity = p_id)
```

optional

$$b = \prod_{k=i}^j a_k$$

■ Further optional arguments:

- DIM, MASK (see later)
- ORDERED: logical value, enforces order of operations

■ Programmer-supplied parts:

```
module mod_p  
  type :: p  
  :  
end type  
type(p) :: p_id = ... ! one  
contains  
  pure type(p) function &  
    prod_p(x, y)  
    type(p), intent(in) :: x,y  
    : ! evaluate product  
end function prod_p  
end module mod_p
```

- operation must be an associative PURE function of exactly two scalar arguments
- identity element covers the case of a zero-sized argument array

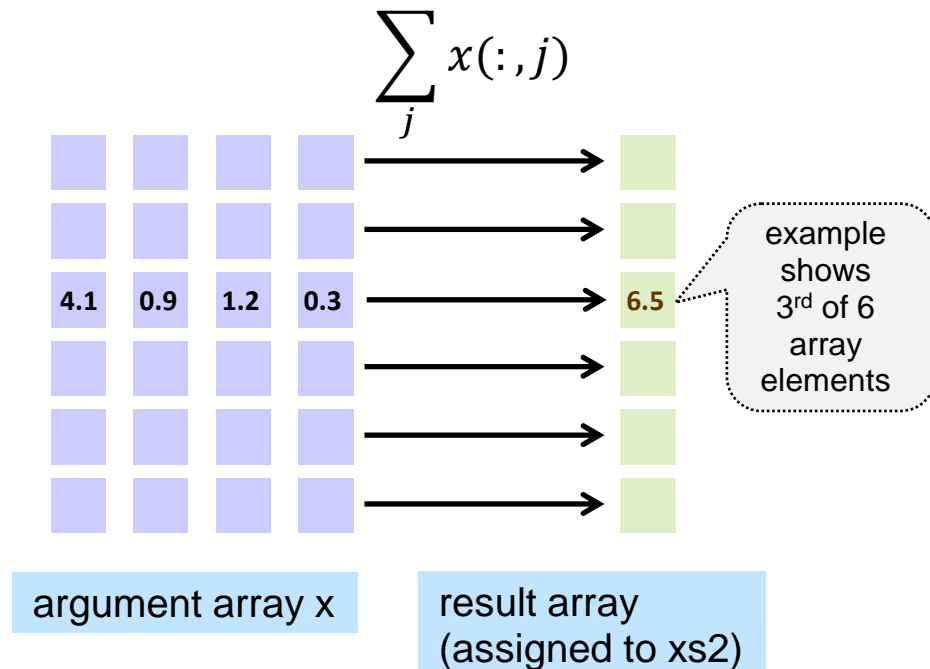
■ Perform reduction along a single array dimension

- other dimensions are treated elementally (\rightarrow result is an **array!**)

```
real :: x(6,4)
real :: xs2(6)
: ! define x
xs2 = sum(x, dim = 2)
```

- example above: `xs2(i)` contains `sum(x(i,:))`
- `dim` must be second argument and/or specified by keyword
- $1 \leq \text{dim} \leq \text{rank of array}$

■ Illustration of reduction along a dimension:



■ Select a subset of elements

- some functions may use a **logical** array **mask** as a third optional argument
- **mask** must have same shape as the first argument

```
real :: a(4), s
a = ...
s = sum(a, mask = a>0.)
```

■ Illustration of masked reduction

4.1 -1. 1.2 0.3 → 5.6

argument array a

result scalar s

■ Combining **dim** and **mask**

- is possible
- both are applied to the first (array) argument

■ Further intrinsics that support **dim** and **mask** exist

- see compiler documentation

Array location intrinsics

Name and arguments	Description
<code>maxloc (array [,dim] [,mask] [,back])</code>	Location of maximum value of an integer or real array
<code>minloc (array [,dim] [,mask] [,back])</code>	Location of minimum value of an integer or real array
<code>findloc (array, value, [,dim] [,mask] [,back])</code>	Location of supplied value in an array of intrinsic type

F08

Logical argument `back`:

- if supplied with value `.false.`, the last identified location is returned
- default value is `.true.`
- added in F08

Example:

```
integer :: x(2,-1:1)
x = reshape([2,3,5,1,1,1], &
            shape=[2,3])
write(*,*) maxloc(x)
write(*,*) maxloc(x, dim=2)
```

1 2

2 1

lower bounds
remapped

```
  1  2  3
1  2  5  1
2  3  1  1
  1  2  3
1  2  5  1
2  3  1  1
```

```
x  -1  0  1
1  2  5  1
2  3  1  1
```

array element
values

Transformational array intrinsics

Name and arguments	Description
<code>dot_product (vector_a, vector_b)</code>	dot (scalar) product of numerical or logical rank 1 arrays.
<code>matmul (matrix_a, matrix_b)</code>	matrix multiplication of numeric arrays of rank 1 or 2
<code>transpose (matrix)</code>	transposition of rank 2 array representing a matrix $\text{matrix}(i, j) \rightarrow \text{matrix}(j, i)$
<code>merge (tsource, fsource, mask)</code>	elemental merging of two arrays of same type and shape, based on logical mask value
<code>spread (source, dim, ncopies)</code>	replicate an array ncopies time along dimension dim
<code>reshape (source, shape [, pad] [, order])</code>	reshape optional arguments: <ul style="list-style-type: none">• pad array to fill in excess elements of result• subscript permutation via integer permutation array order
<code>cshift (array, shift [, dim])</code>	circular shift of array elements along dimension 1 or dim
<code>eoshift (array, shift [, boundary] [, dim])</code>	end-off shift of array elements along dimension 1 or dim , using boundary to fill in gaps if supplied

Array intrinsics: Packing and unpacking

Transformational functions:

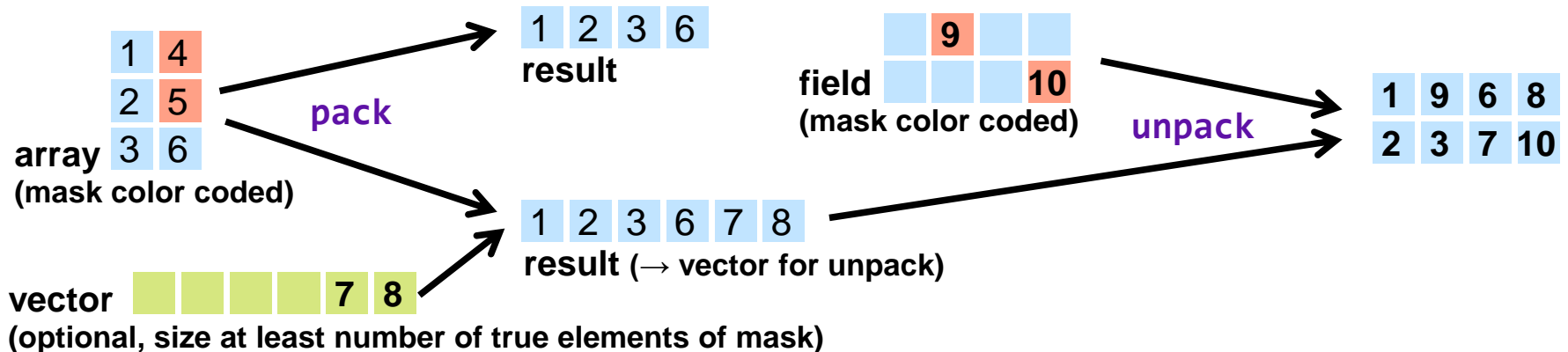
- convert from multi-rank arrays (of any type) to rank 1 arrays (of same type) and back
- a logical mask is used to select a subset of array elements (may be a scalar with value `.true.`)

```
pack (array, mask [, vector])
```

```
unpack (vector, mask, field)
```

Unpack result:

- type is that of `vector`
- shape is that of logical array `mask`
- size of `vector`: at least number of true elements of `mask`
- `field` of same type as `vector`, and a scalar, or same shape as `mask`





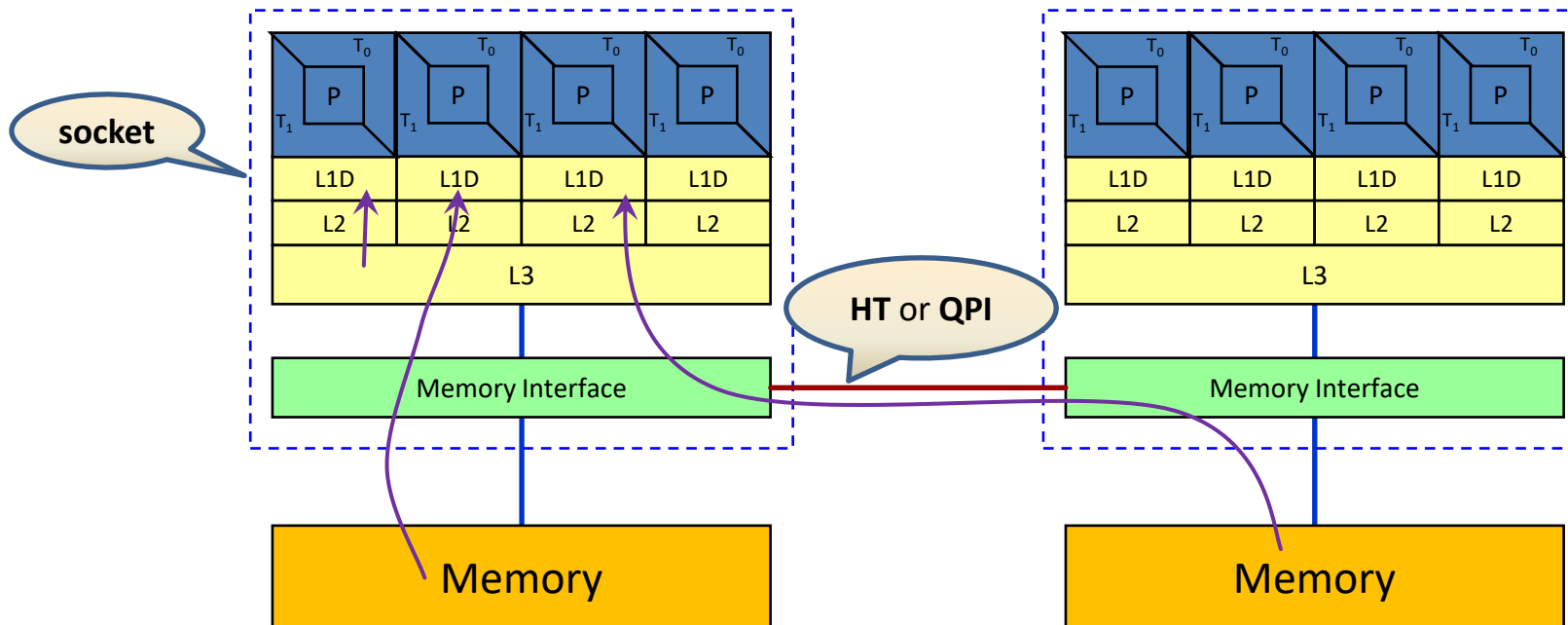
Performance of serial code

Standard Architectures of this decade

- **multi-core** **multi-threaded** processors with a deep cache hierarchy

Illustration shows 4 cores per socket. Typical: 8 – 14 cores

- typically, two **sockets** per node



ccNUMA architecture: „cache-coherent **non-uniform** memory access“

Concept of cache

■ A small but fast memory area

- used for storing a (small) memory working set for efficient access

■ Reasons:

- physical and economic limitations

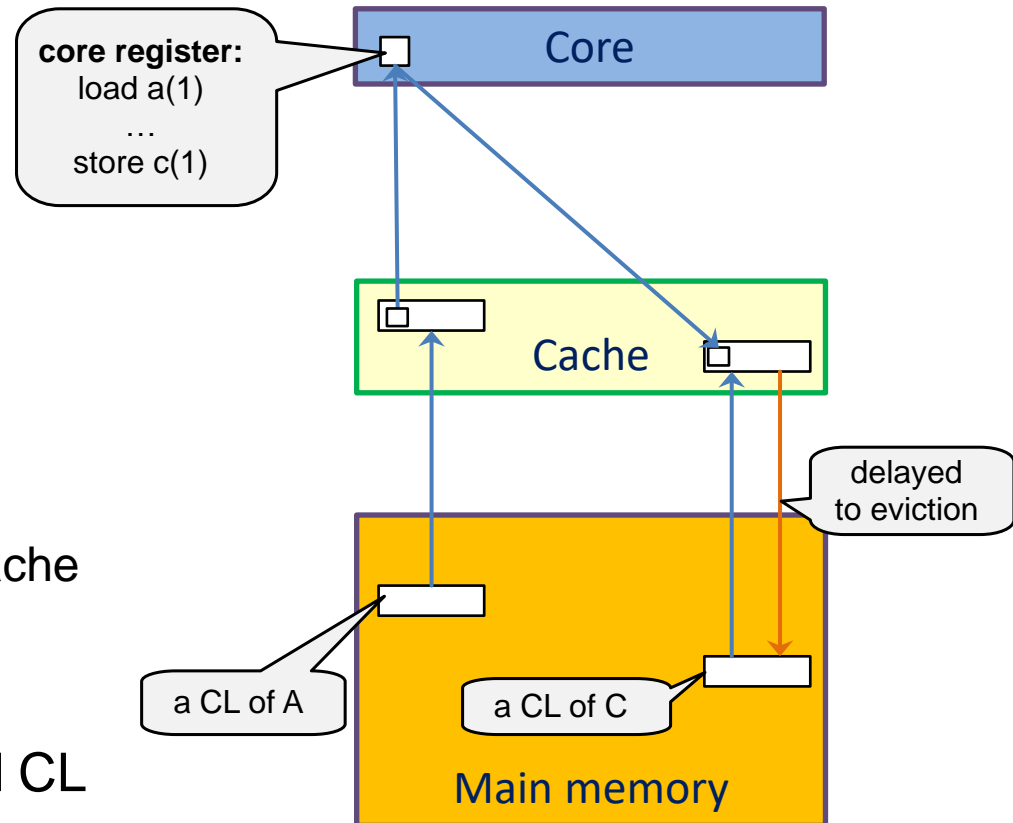
■ Loads/Stores to core registers

- may trigger cache miss \rightarrow transfer of memory block („cache line“, CL) from memory

■ Cache fills up ...

- usually least recently used CL is evicted

■ Example: $c(:) = a(:) + \dots$



■ This course

- limits itself (mostly) to **serially** executed code
- only **one core** of a node is used

■ For efficient exploitation of the architecture

- you need to enable use of **all** available resources

■ Possible execution modes:

- **throughput** – execute multiple instances of serial code on a single node (parameter study)
- **capability** – enable parallel execution of a single instance of the program

which to use depends on the resource needs vs. their availability

■ Parallel models

- inside Fortran:
DO CONCURRENT
Coarrays
- outside Fortran:
Library approach (MPI)
Directive approach (OpenMP, OpenACC)

briefly touched in this course

■ Conceptual scalability

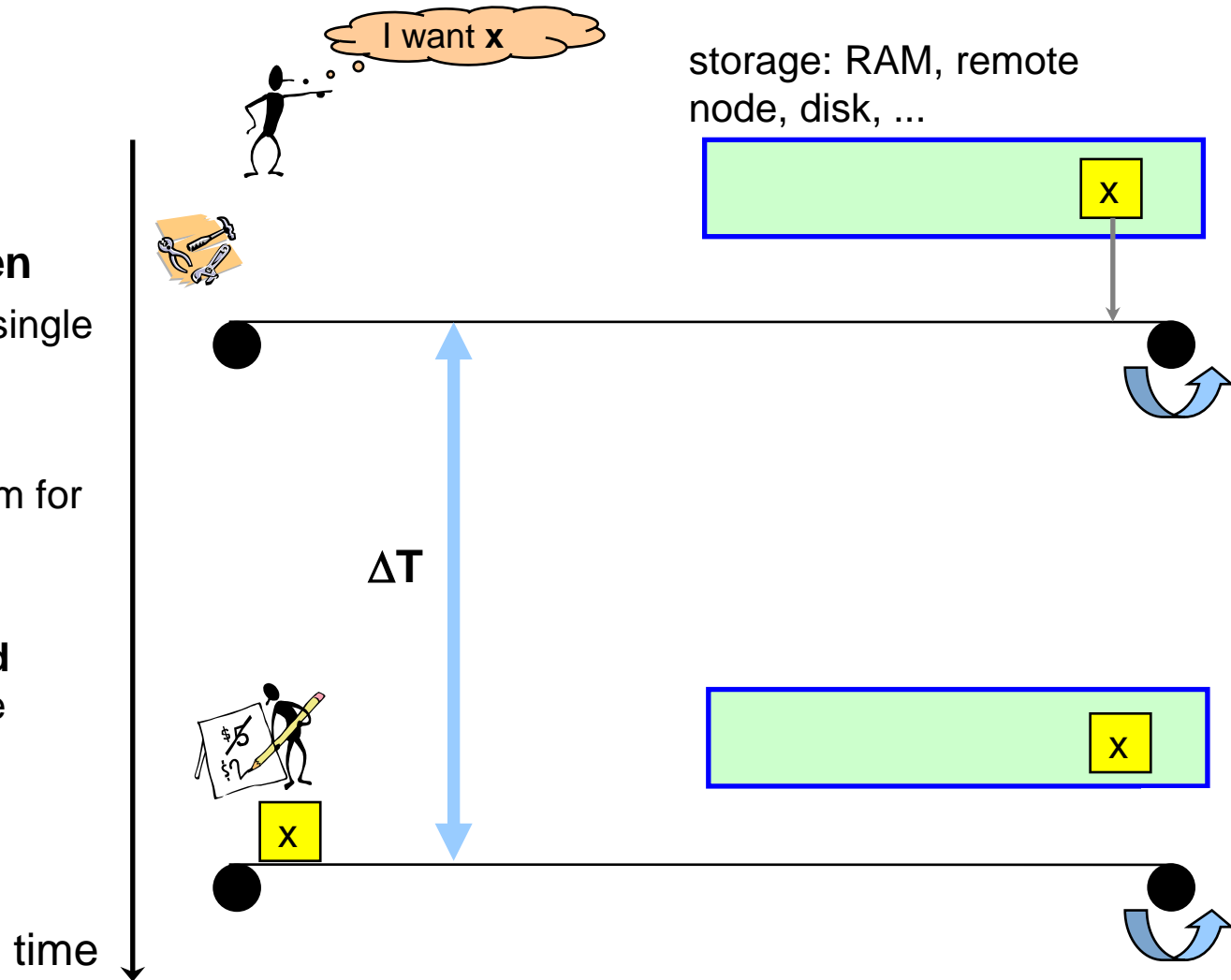
- **shared memory:**
program execution limited to a single node
- **distributed memory:**
ability of program to execute on multiple nodes, and exchange data between them

Two very important words from the HPC glossary

Latency

Time interval ΔT between

- **request** of worker for single datum
- and
- **availability** of data item for being worked on
- depends on speed **and** length of assembly line

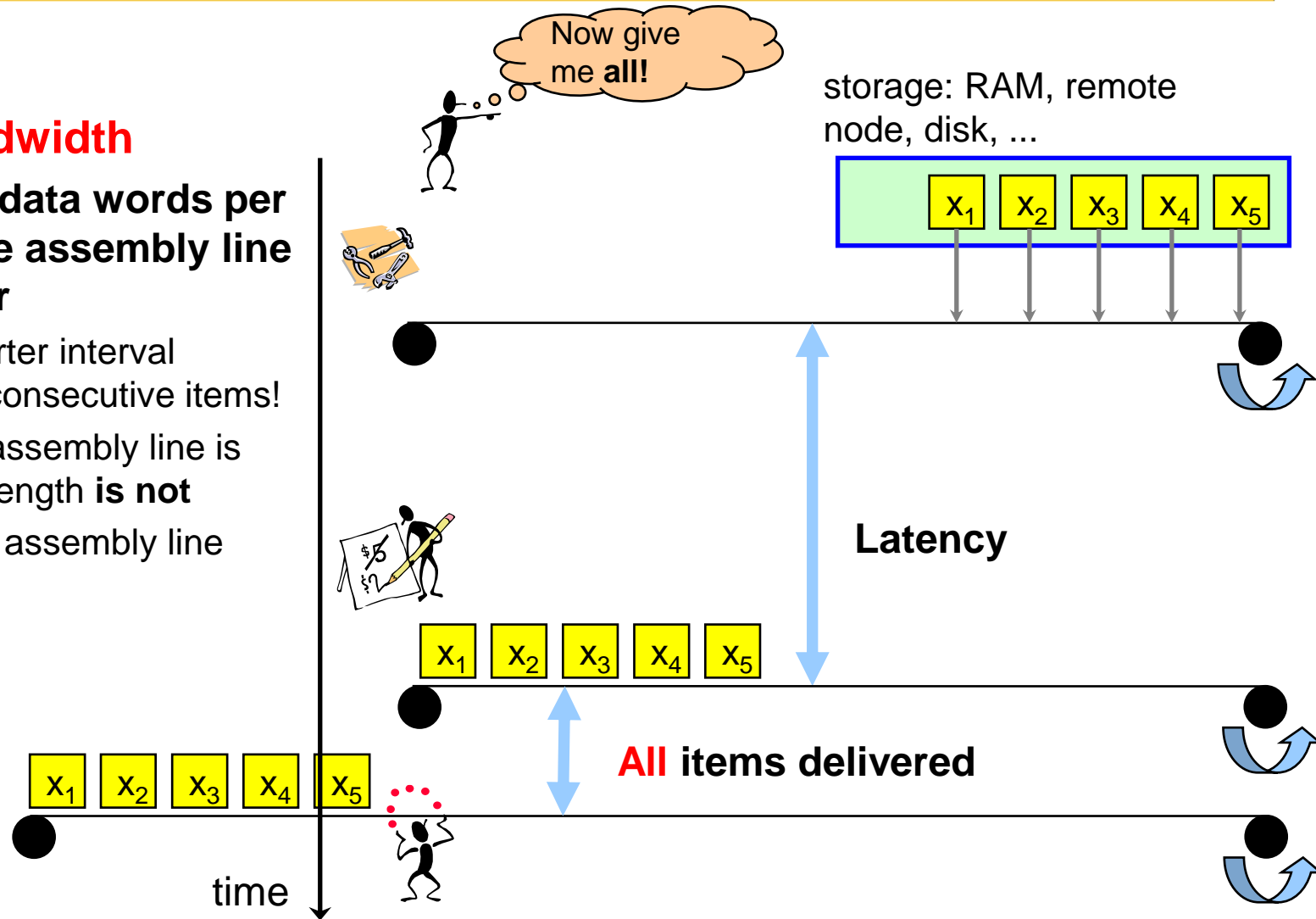


... and here the second one

Bandwidth

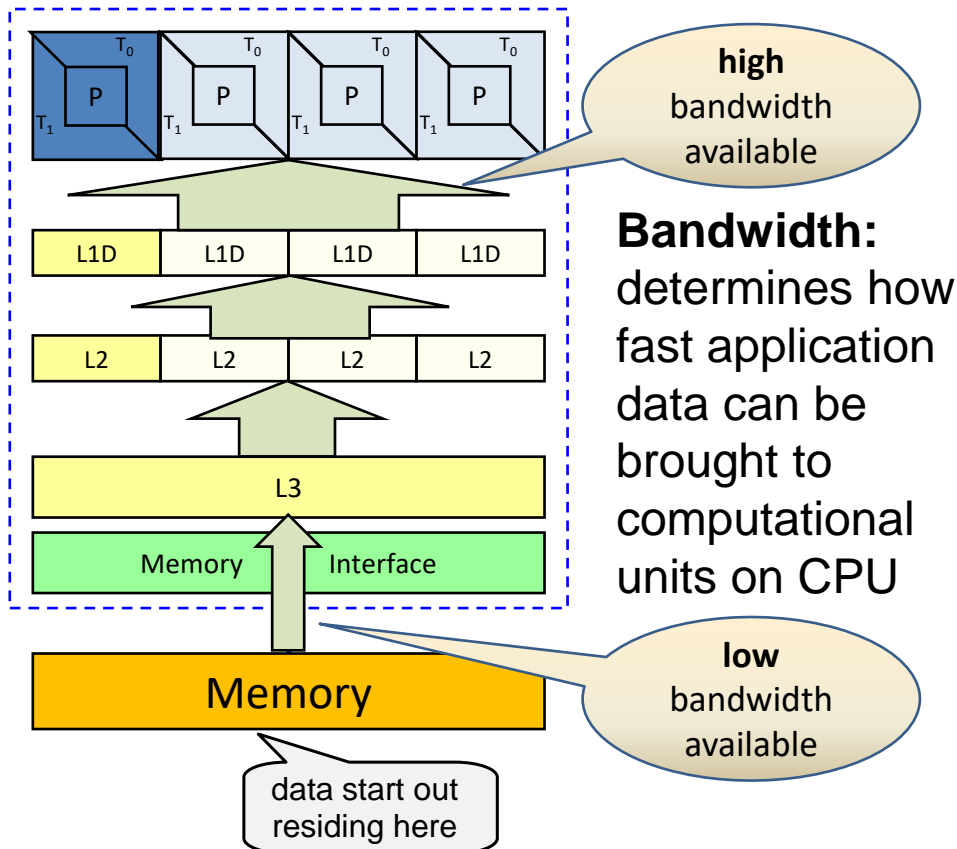
Number of data words per second the assembly line can deliver

- much shorter interval between consecutive items!
- speed of assembly line is relevant, length **is not**
- aim: keep assembly line **full!**



■ Performance Characteristics

- determined by memory hierarchy



■ 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

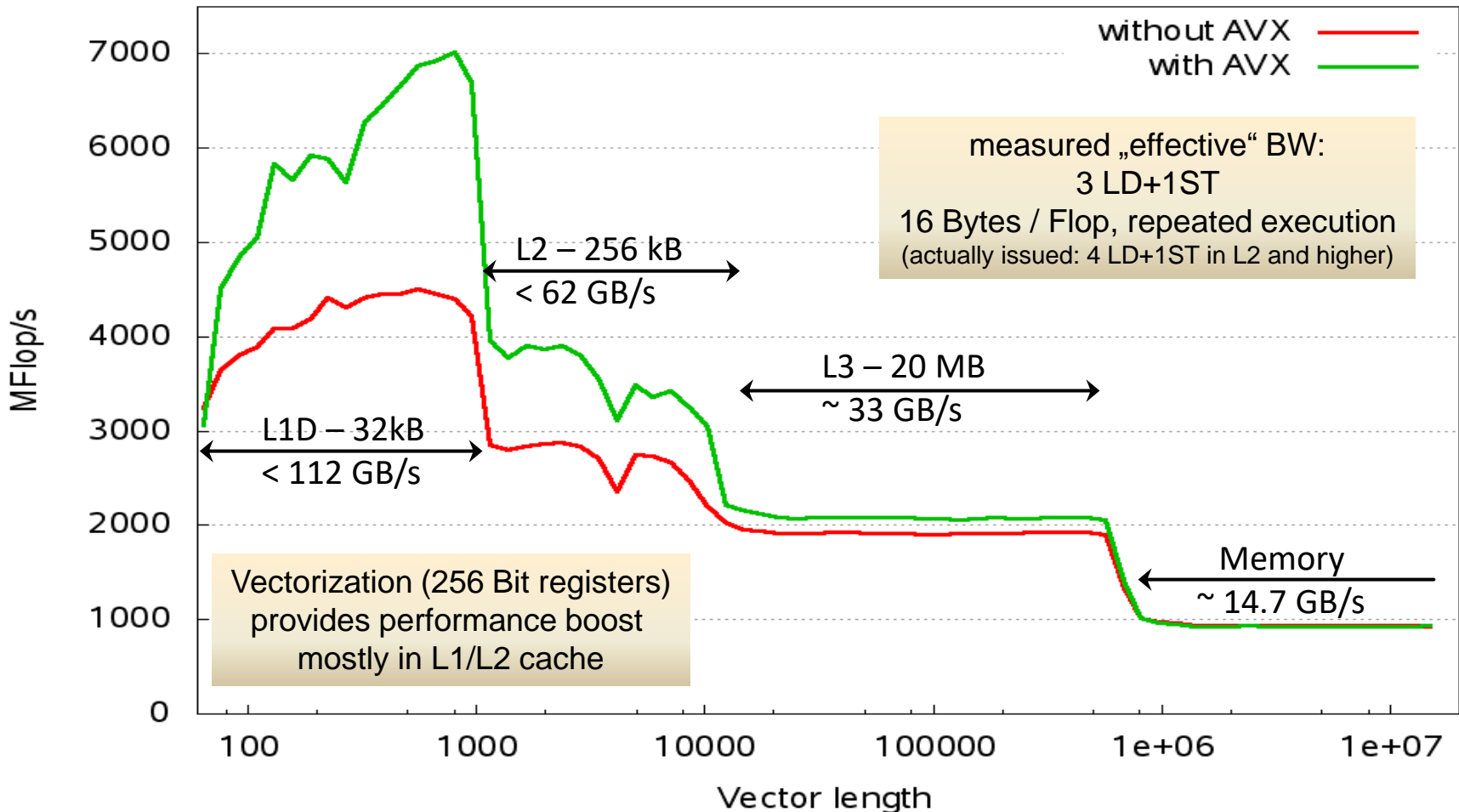
- known operation count, load/store count
- some variants of interest:

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)

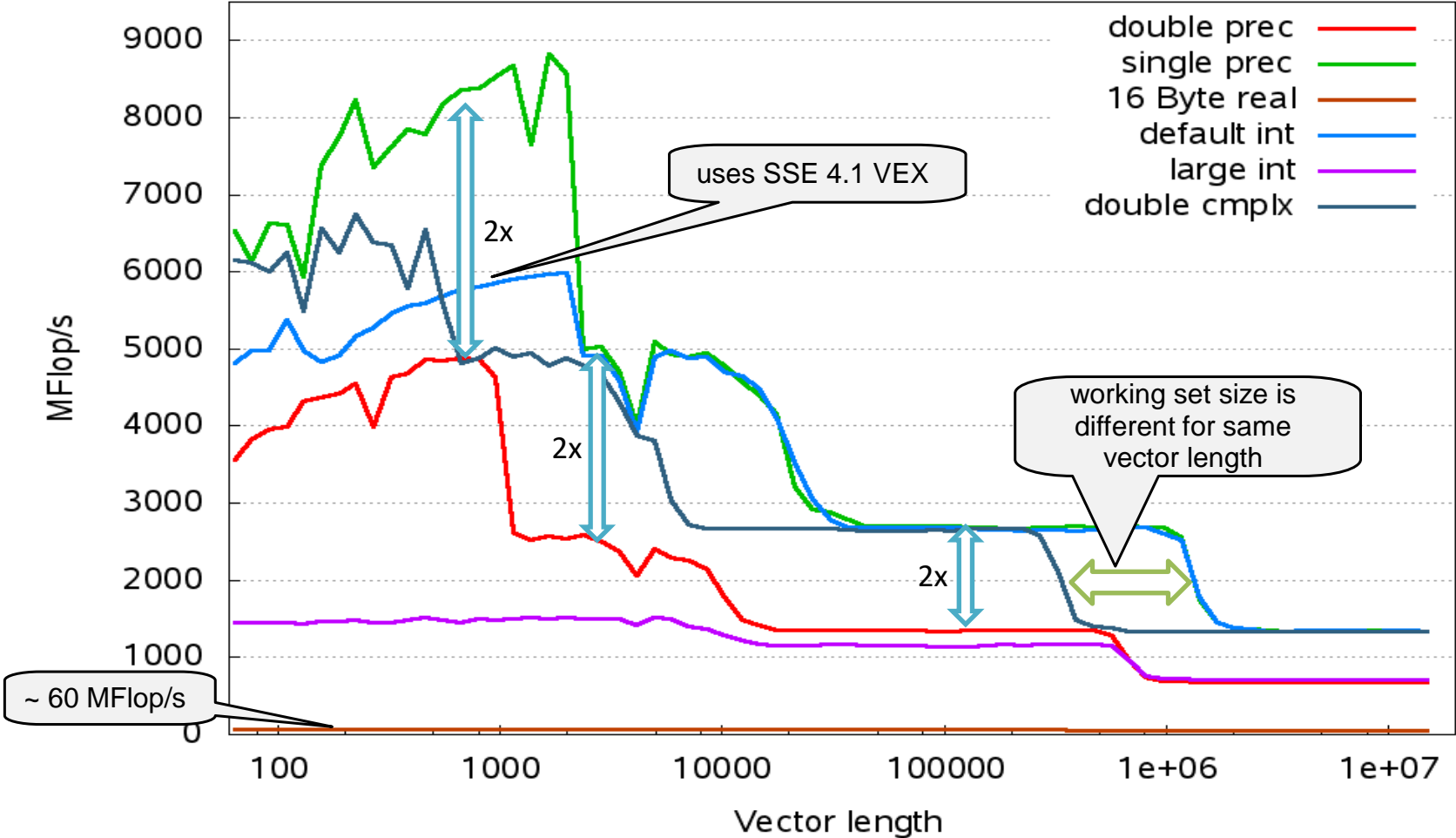
Vector Triad $D(:) = A(:) + B(:) * C(:)$

- **Synthetic benchmark:** bandwidths of „raw“ architecture, looped version for a **single core** Sandy Bridge 2.7 GHz / ifort 13.1



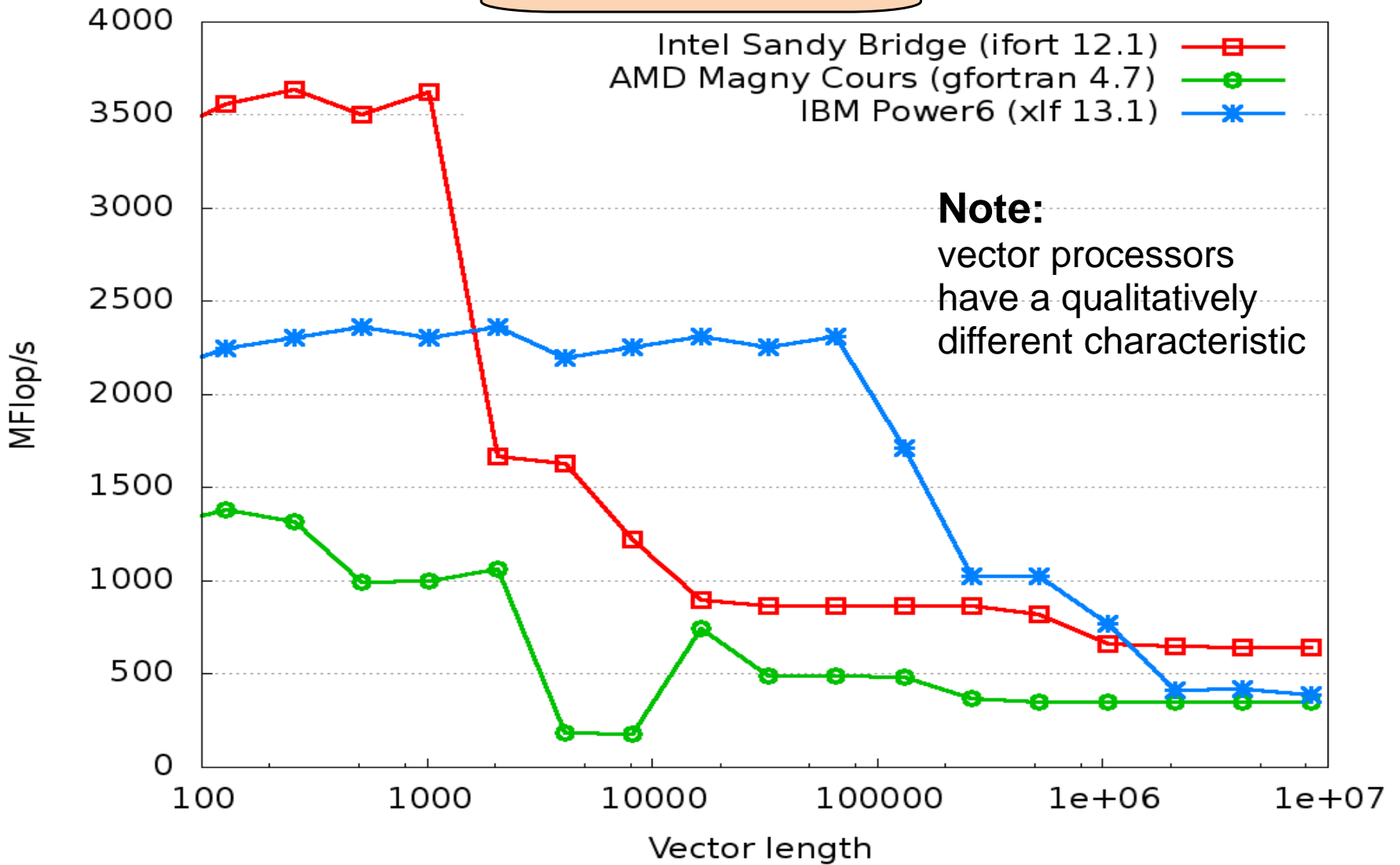
Performance by type and kind

Sandy Bridge 2.3 GHz with AVX / ifort 16.0



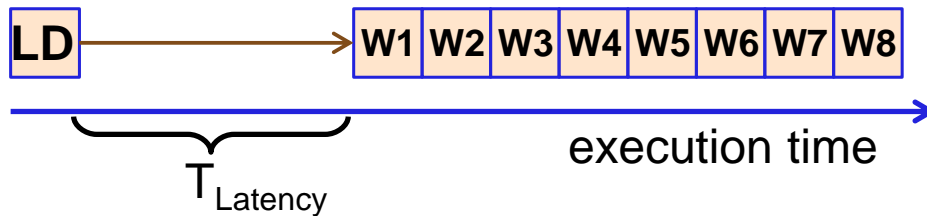
Hardware dependence of Triad Performance

Double Precision Triad



■ Loads and Stores

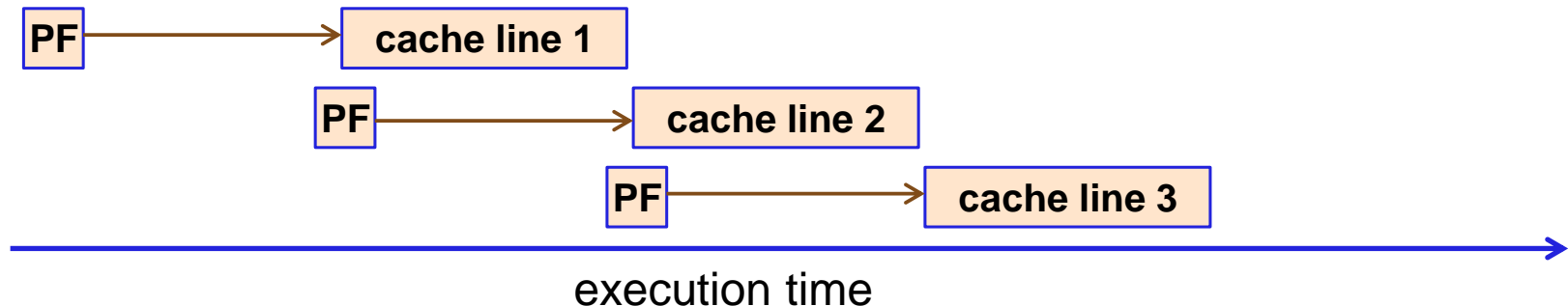
- apply to cache lines



- 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-Requirement:

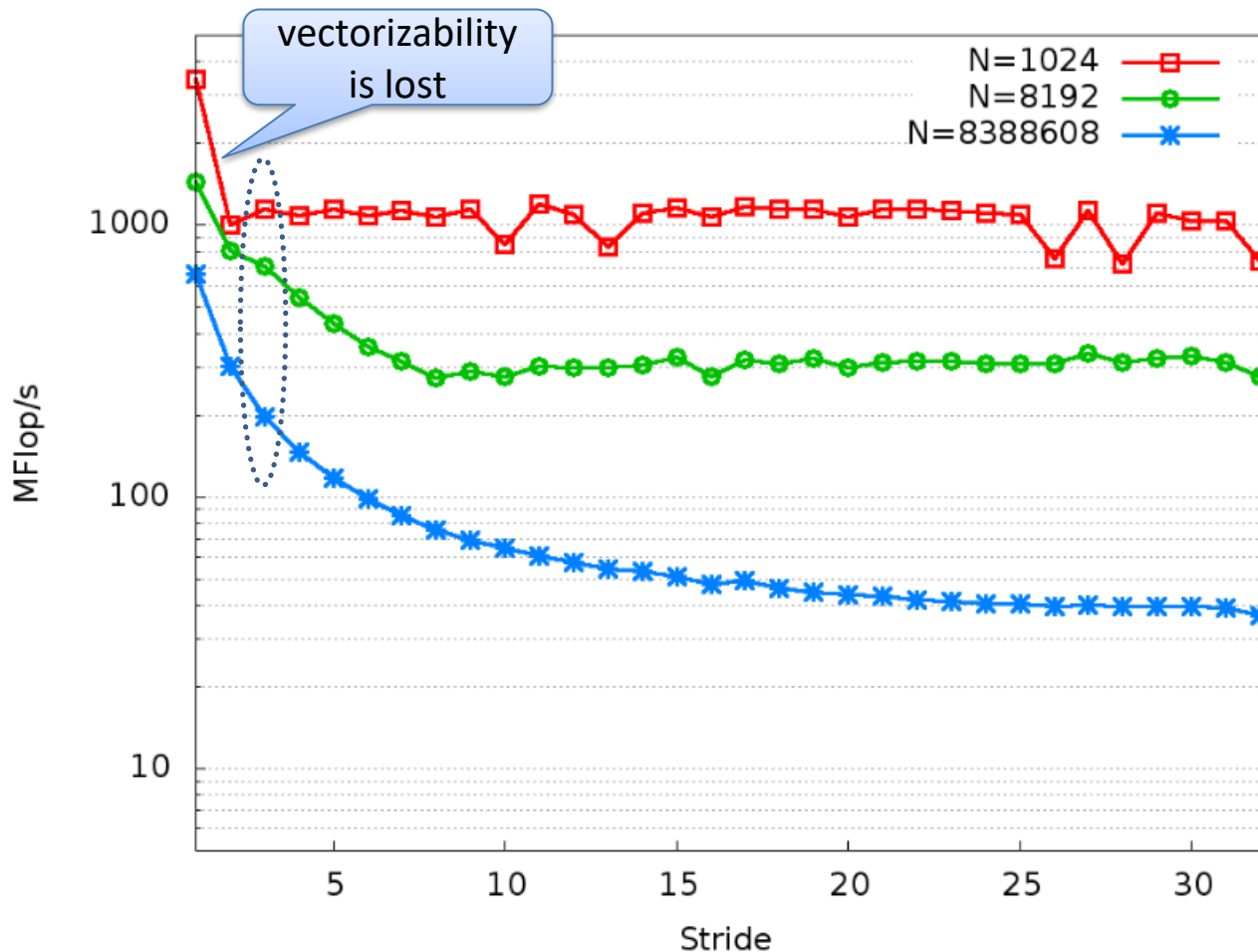
- **spatial** locality
- **violation** of spatial locality:

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)

$$D(::\text{stride}) = A(::\text{stride}) + B(::\text{stride}) * C(::\text{stride})$$

Example: stride 3



Notes:

- stride known at compile time
- serial compiler optimizations may compensate performance 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

```
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

- **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 (discussed previously)
- **With Fortran 90 and later**
 - extension of the existing rules was necessary (not discussed in this course)
- **Other languages have caught up**
 - e.g. beginning with C99, C has the `restrict` keyword for pointers → similar aliasing rules as for Fortran
 - also, compiler aliasing analysis has improved



After the Lunch break ...

Fortran Environment

■ Processing the command line

<code>command_argument_count()</code>	integer function that returns what it says
<code>get_command_argument(number [, value] [, length] [, status])</code>	subroutine that delivers information about a single command line argument
<code>get_command(command [, length] [, status])</code>	subroutine that delivers information about the complete command line

■ Executing system commands

<code>execute_command_line(command [, wait] [, exitstat], [, cmdstat] [, cmdmsg])</code>	subroutine that executes a system command specified as a string Replaces the non-standard extension <code>call system(command)</code>
--	--

■ Process environment variables

<code>get_environment_variable(name [, value] [, length] [, status] [, trim_name])</code>	subroutine that delivers information about a named environment variable
---	---

■ Obtain the value of the PATH variable:

```
integer, parameter :: strmx = 1024
character(len=strmx) :: path_value
integer :: path_length, istat
call get_environment_variable('PATH', length=path_length, &
                             status=istat)

if (istat /= 0) &
  stop 'PATH undefined or environment extraction unsupported'
if (path_length > strmx) &
  write(*, *) 'Warning: value of PATH is truncated'

call get_environment_variable('PATH', path_value)
```

■ These intrinsics support additional diagnostics

- it is strongly recommended to use them
- see intrinsics documentation for details

- Contains some often-used constants

- Here a subset:

Name	Purpose
<code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code>	integer KINDs by size in bits
<code>real32</code> , <code>real64</code> , <code>real128</code>	real KINDs by size in bits
<code>integer_kinds</code> , <code>real_kinds</code> , <code>character_kinds</code> , <code>logical_kinds</code>	constant arrays containing all supported KIND numbers
<code>character_storage_size</code> <code>numeric_storage_size</code> <code>file_storage_size</code>	storage sizes in bits

- Contains some inquiry procedures

```
compiler_options()
```

```
compiler_version()
```

- return string constants

- Some of this was added in F08

<https://gcc.gnu.org/onlinedocs/gfortran/Intrinsic-Modules.html>

```
use, intrinsic :: iso_fortran_env
implicit none
integer, parameter :: wp = real64, ik = int32, strmx=128

real(kind=wp) :: x
integer(kind=ik) :: i4

character(len=strmx), parameter :: version = compiler_version()
```

■ Additional INTRINSIC keyword on USE statement

- use of this is recommended to avoid mistaken access to a non-intrinsic module with the same name

■ Comment on KIND numbers

- declarations like REAL*8 (using **byte** units) are supported in many compilers, but are **not** standard-conforming



Scoping and Lifetime of objects

Derived types and interfaces

```
module mod_scoping_1
  implicit none
  type :: p
    integer :: i
    real :: x
  end type
  real :: x
  abstract interface
    real function f(x, s)
      import :: p
      real, intent(in) :: x
      type(p), intent(in) :: s
    end function
  end interface
end module
```

scope 2

scope 3

scope 1

type components are „class 2“ identifiers; must be unique per-type

no collision of global variable with type component or dummy argument in interface

interface has no host access
→ type definition must be IMPORTed

F03

Examples for nested scoping (2)

Global and local variables; host association

```

module mod_scoping_2
  implicit none
  integer :: is, js, ks
  contains

```

```

  subroutine proc(is)
    integer, intent(in) :: is
    integer js
    js = is
    ks = ifun()

```

```

  contains

```

```

    integer function ifun()
      ifun = is + js
    end function

```

```

  end subroutine

```

```

end module

```

dummy **is** is a „class 3“ identifier
(must be unique per-interface);
it is a separate entity that overrides global **is**

local **js** in scope 2;
it is a separate entity that overrides global **js**

ks is host associated from scope 1

is and **js** are host associated from scope 2



source of programming errors or performance problems

- forget local declaration for entities meant to be local → access by host association
- `implicit none` does not help here
- less probable if suggestive names are given to globals

■ **IMPORT statement: extended in** F18

```
module mod_scoping_2
  implicit none
  integer :: is, js, ks
  contains
  subroutine proc(is)
    import, only : ks
    integer, intent(in) :: is
    integer js
    js = is
    ks = ifun()
    contains
    integer function ifun()
      import, all
      ifun = is + js
    end function
  end subroutine
end module
```

scope 3

scope 2

scope 1

■ **Variants:**

- **IMPORT, ALL** makes all entities from the host available.
- **IMPORT, ONLY : ...** makes a set of variables from the host accessible, all others require declaration (or follow implicit typing rules)
- **IMPORT, NONE:** no variable from the host is accessible.

■ Typical situation:

- (memory for an) entity exists from start of execution of its scoping unit
- until execution of its scoping unit has completed

■ Definition status:

- may become defined after start of execution, or not at all
- will become undefined once execution of its scoping unit completes

■ Exceptional situation:

- module variables („globals“) are persistent
(static module variables exist for the duration of the program execution)
- Fortran terminology: they **implicitly** have the SAVE attribute
- disadvantage for shared-memory parallelism: not thread-safe

■ Example

- (legacy) standalone procedure

```
subroutine process(a, n)
  implicit none
  real :: a(*)
  integer :: n
```

attribute form

```
integer, save :: first = 0
real :: work(1000)
save :: work
```

statement form

```
if (first .EQ. 0) then
```

```
  ...
  work(...) = ...
  first = 1
```

expensive calculation of reusable array `work` is done once only


```
end if
```


```
  :
end subroutine
```

update array `a`

■ Properties:

- at the subsequent invocation of the procedure, SAVEd local variables have the same definition status and value as at the end of the previous execution
→ „lifetime extension“

 for recursive subroutines, only **one instance** of SAVEd local variable exists for all active invocations

 a blanket SAVE statement applies to all local variables in the scoping unit, or all module variables if in the specification section of a module
→ avoid the above two items



Constant Expressions, Initializations, and Specification Expressions

- **Statements which provide initial values to**
 - named constants
 - variables (module or local)
 - data in COMMON blocks (not treated in this course)
- **The actual values must be specified as **constant expressions****
 - rules allow to perform all initializations at **compile time**
 - historical note: constant expressions were earlier known as **initialization** expressions

■ Intent:

- provide a value with which a (local or global) variable is defined at the beginning of the first execution of its scoping unit

■ Variant 1:

```
integer :: i = 5
character(len=4) :: cn='f'
type(date) :: o = date(...)
real :: xx(3) = [ 0.,0.,0. ]
```

- follow the declaration with a **constant expression**
- rules as for intrinsic assignment

■ Variant 2: the DATA statement

```
integer :: i
character(len=4) :: cn
type(date) :: o
real :: xx(3)
data i, o / 5, date(...) /, &
      cn, xx / 'f ', 3*0.0 /
```

- sequence of values matching the type of each element of the object list
- note the repeat factor for the array initial values

■ Recommendation:

- variant 1 for readability

Consequences:

- initialized variables acquire the (implicit) SAVE attribute
- **different** from C semantics (similar syntax!)

```
subroutine f_proc()  
  integer :: i = 0  
  ... = i + ...  
end subroutine
```

value 0 only on first invocation, then that from last invocation

```
void c_proc() {  
  int i = 0;  
  ... = i + ...  
}
```

value 0 always.
`int i=0;` is an executable stmt and `i` is not a static variable

```
subroutine f_proc2()  
  integer :: i  
  i = 0  
  ... = i + ...  
end subroutine
```

same semantics as C code above

Constant expressions:

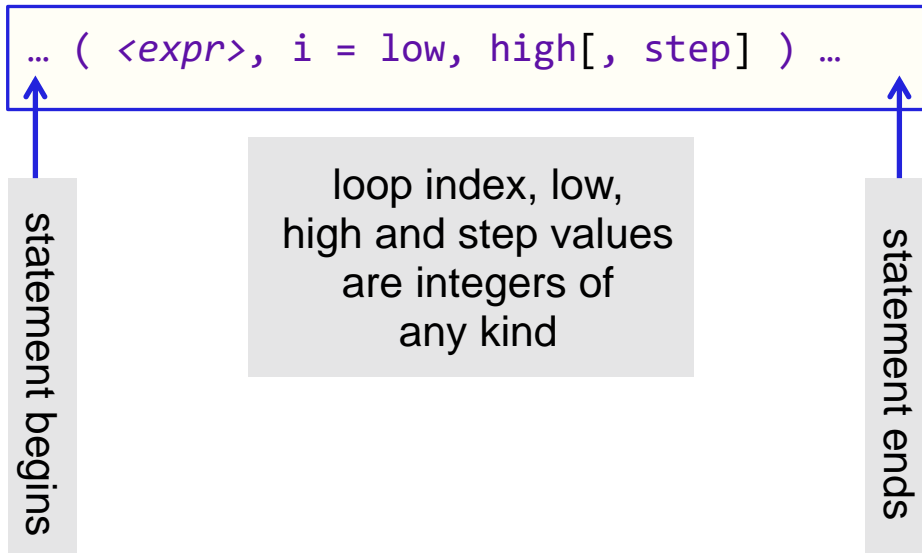
- built from constants, subobjects of constants and other constant expressions
- may use array or structure constructors
- may use intrinsic operations
- **certain intrinsic functions:** elemental intrinsics, array inquiry functions, transformational intrinsic functions
- **Note:** F95 was more restrictive with respect to which intrinsics were allowed; `**` could be used only with integer exponent.

The above list is not entirely complete

Implied-do loops (1)

■ Within-statement processing of array expressions

- need to generate a local scope for loop index



- may be nested → must then use **distinct** iteration variables

■ Three scenarios:

1. constant expression within a DATA statement
2. within an array constructor (not necessarily a constant expression)
3. within an I/O data transfer statement
→ will be treated in context of I/O (not a constant expression)

■ Examples for scenario 1

```
real :: a(10), b(5,10)
integer :: i, j

data (a(i), i=2,4) / 1.0, 2.0, 3.0 /
data ((b(i, j), i=1,5), j=1,10,2) / 20*0.0, 5*1.0 /
```

- both DATA statements perform **partial** initialization:
a(2:4) and b(:, ::2) are initialized
- initialization of b uses two nested implied-do loops

■ Examples for scenario 2

```
integer :: i, j

real :: aa(10) = [ ( real(i),i=1,size(aa) ) ]
real :: bb(5,10)

bb = reshape( [ ( ( sin(real(i)), i=1,size(bb,1) ),
                  j=1,size(bb,2) ) ], shape(bb) )
```

- for `bb`, a rank-1 array is constructed via two nested implied-do loops, then `reshape()` is used to convert to a rank-2 array
- if the complete implied-DO loop is intended to be a constant expression, the argument expression must be a constant expression

■ Specify default values for derived type components

- at component declaration inside type definition

```
module mod_person
  use mod_date
  implicit none
  type, public :: person
  private
  character(len=smx) :: name = 'Unknown'
  type(date) ←: birthday
  character(len=smx), public :: location
end type
end module
```

```
type :: date
  private
  integer :: year = 0, &
           mon = 0, day = 0
end type
```

constant expression
required

- need not do so for all components (in fact it may not be possible for components of opaque type)
- derived type components: any pre-existing initialization is overridden if a default initialization is specified

■ Objects of such a type:

- components are **default initialized** to values specified in type definition

```
type(person) :: chairman
```

```
write(*,*) chairman % name
```

```
write(*,*) chairman % birthday
```

```
chairman = person(location = 'Room 23')
```

prints the two lines:

Unknown

0 0 0

default initialized components
can be omitted from constructor

■ Further properties of default initialization

- can be overridden by explicit initialization (DATA **disallowed** in this situation)
- applies to static and dynamic objects (including automatic objects, local variables, function results – see later); is independent of component accessibility
- does **not** by itself imply the SAVE attribute for objects
- INTENT(OUT) objects of such a type: are default initialized upon invocation of the procedure

Specification expressions: Providing data needed for specifications

■ A special class of expressions:

- may need to be evaluated on entry to a procedure at beginning of its execution (i.e., **run time evaluation**)
- can be used to determine array bounds and character lengths in specification statements → these are **integer valued scalars**

■ Inside a specification expression

- a restricted form of non-integer expressions can occur

■ Restricted expressions:

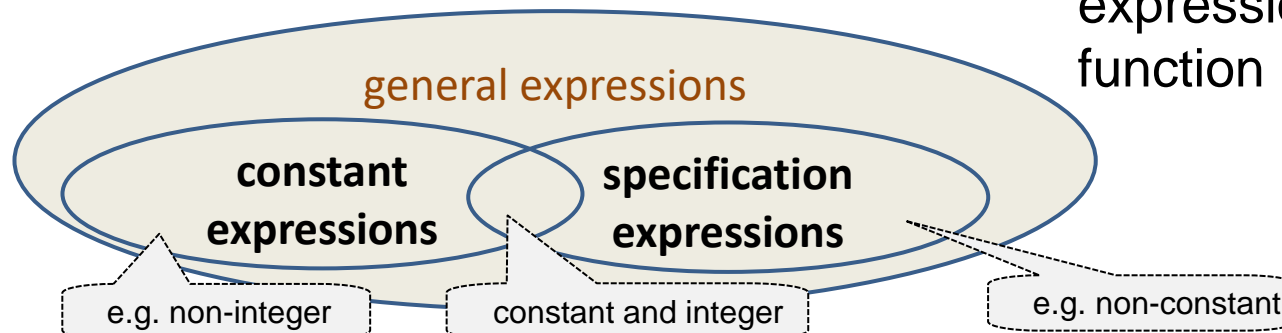
- built from constants, subobjects of constants, dummy arguments, host variables or global entity object designators (with some restrictions) and other restricted expressions
- intrinsic functions, **specification inquiries or specification functions**
- intrinsic operations
- array or structure constructors, implied-do

■ Subclass of inquiry intrinsics e.g.,

- array inquiry function `size()`, ...
- bit inquiry function `bit_size()`
- character inquiry `len()`
- numeric inquiry `huge()`, ...
- type parameter inquiry

■ Subclass of **user-defined** functions

- must be PURE
- must not be internal, or a statement function
- must not have a procedure argument
- must have an explicit interface
- **Note:** a recursive reference in a specification expression inside such a function is not allowed




Function returning a string

```
function pad_string(c, n) result(s)
  character(len=*) :: c
  integer :: n
  character(len=len(c)+n) :: s
  :
end function
```

explicit interface
required

Not permitted:

- non-constant expression in main program or module spec. part



```
program p
  integer :: n = 7
  real :: a(2*n)
  :
end program
```

→ compiler throws error

Declare working space

- automatic** (non-SAVED!) variables

```
module mod_proc
  integer, parameter :: dm = 3, &
                        da = 12
contains
  subroutine proc(a, n)
    real a(*)
    integer :: n
    real wk1( &
              int(log(real(n))/log(10.)) )
    real wk2( sfun(n) )
    :
    wk1, wk2 removed at end of procedure
  end subroutine proc
  pure integer function sfun(n)
    integer, intent(in) :: n
    sfun = dm * n + da
  end function sfun
end module mod_proc
```

restricted expression

specification function

wk1, wk2 removed at end of procedure

■ A special-case variant of dynamic memory

- usually placed on the stack
- dynamic memory is otherwise managed on the heap → treated soon

■ 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

Now we proceed to an
exercise session ...



Array Processing – Part 2

Procedure interfaces and block constructs

■ 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:)
```

■ 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`

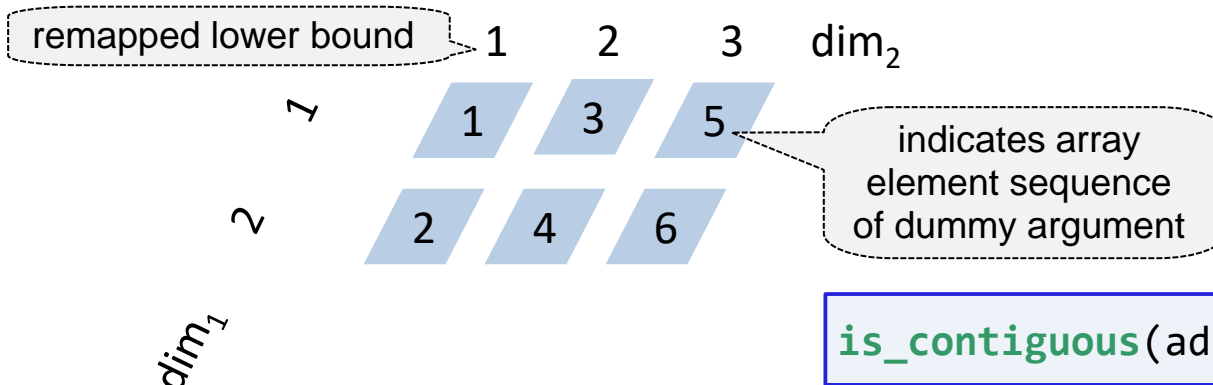
consistency of argument's
type, kind and rank
with interface specification
is **required**

■ Actual argument

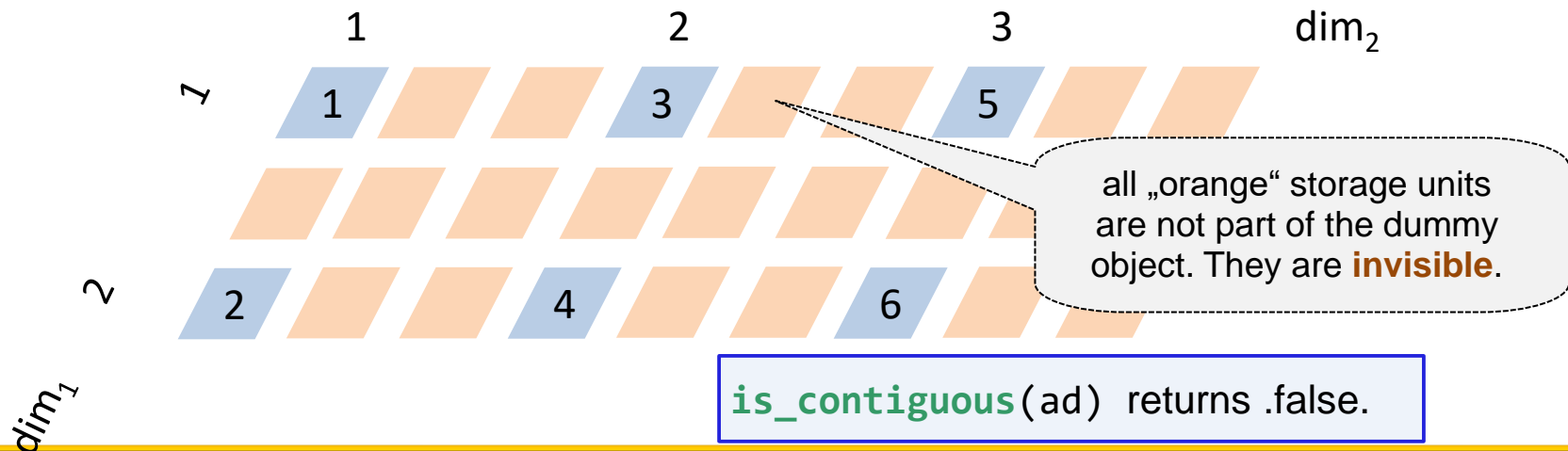
- must have a shape
- can be an array section
- normally, a descriptor will be created and passed → no copying of data happens

Memory layouts for assumed shape dummy objects

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



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



■ For large problem sizes,

- non-contiguous access inefficient due to loss of spatial locality

```
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:

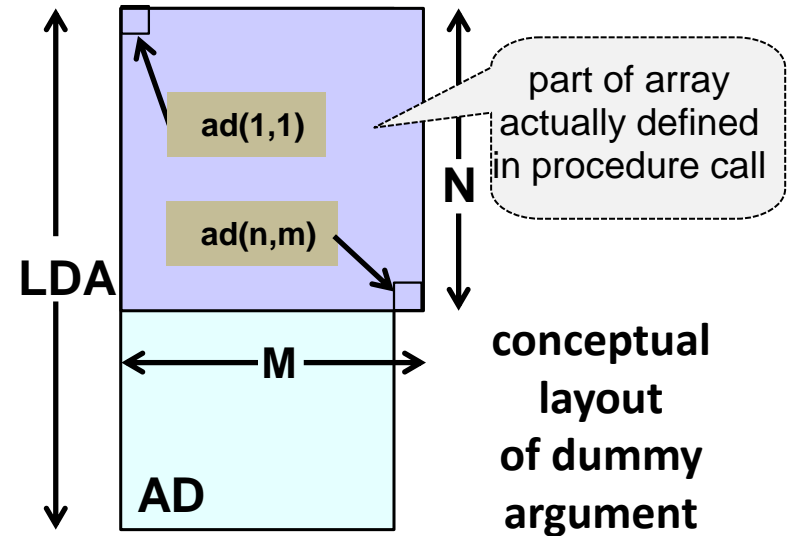
- with a contiguous actual argument → passed as usual
(actual argument: a whole array, a contiguous section of a whole array, or an object with the CONTIGUOUS attribute, ...)
- with a non-contiguous actual argument → copy-in / copy-out
(creating the compactified temporary array has some overhead!)

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

does not have
a shape because
size is assumed
from actual arg.



Notes:

- **leading dimension(s)** of array as well as **problem dimensions** must be explicitly passed
this permits (but does not force) the programmer to assure that $ad(i,j)$ corresponds to element (i,j) of the actual argument
- actual memory requirement implied by addressing: $LDA * (M-1) + N$ array elements
- Example: Level 2 and 3 BLAS interfaces (e.g., DGEMV)

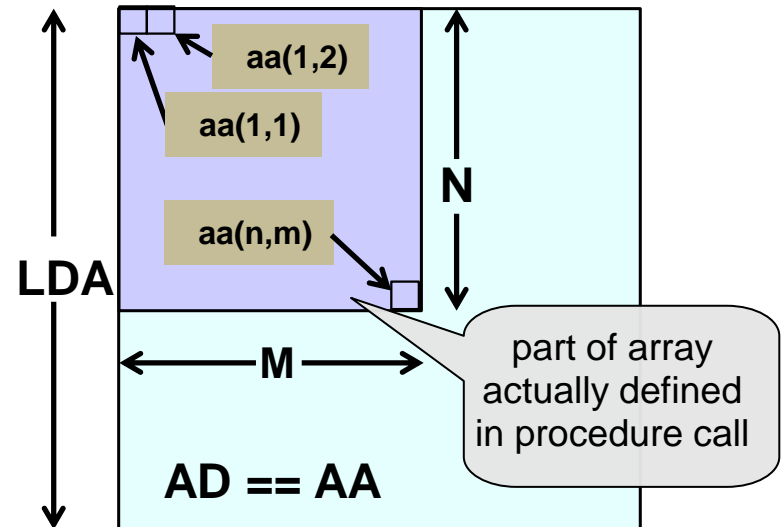
Assumed size: typical usage

Actual argument is

- a complete array
- of same type, kind and rank as dummy argument

```
integer, parameter :: lda = ...
real :: aa(lda, lda)
:
: ! calculate n, m
call slvr( aa, lda, n, m )
```

- behaves as if address of first array element is passed to procedure



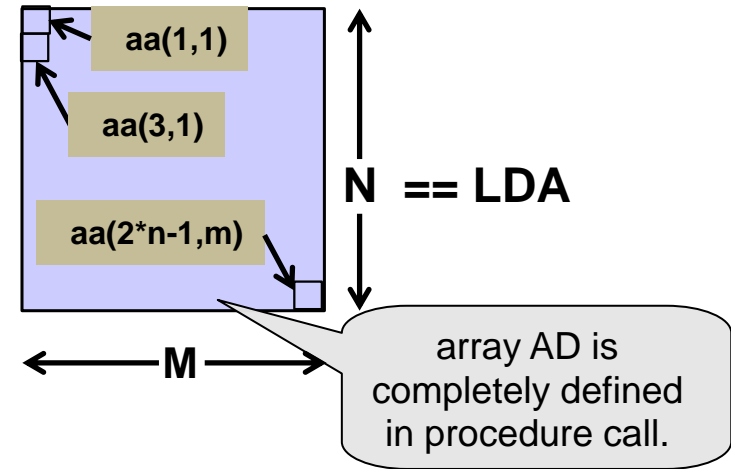
Actual argument is

- a non-contiguous array subobject (selected by sectioning or vector subscripting)
- of same type, kind and rank as dummy argument

for INTENT(IN) only

```
integer, parameter :: lda = ...  
real :: aa(lda, lda)  
:  
: ! calculate n, m  
call slvr( aa(1:2*n:2,:), n, n, m )
```

AD is a compactified copy of AA(1:2*n:2,:)



i.e., size(aa(1:2*n:2,:), 1)

- causes **copy-in/copy-out**: a contiguous temporary array is created and passed to the procedure

Assumed size: rank mismatch

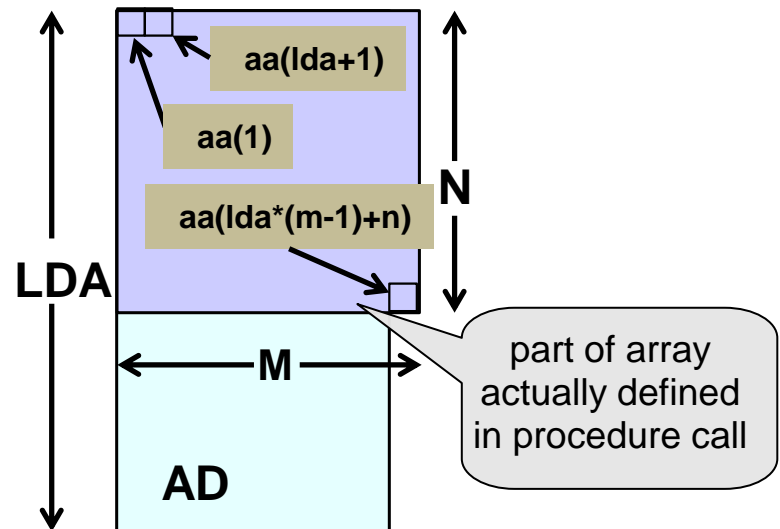
Actual argument is

- a complete array
- of same type and kind as dummy argument
- but of **different** rank

```
integer, parameter :: aadim = ...
real :: aa(aadim)
:
: ! calculate lda, n, m
call slvr( aa, lda, n, m )
```

example:
rank 1

- behaves as if address of first array element is passed to procedure
- data layout must be correctly set up by caller



Assumed size: array element as actual argument

Example:

- blocked processing of subarrays

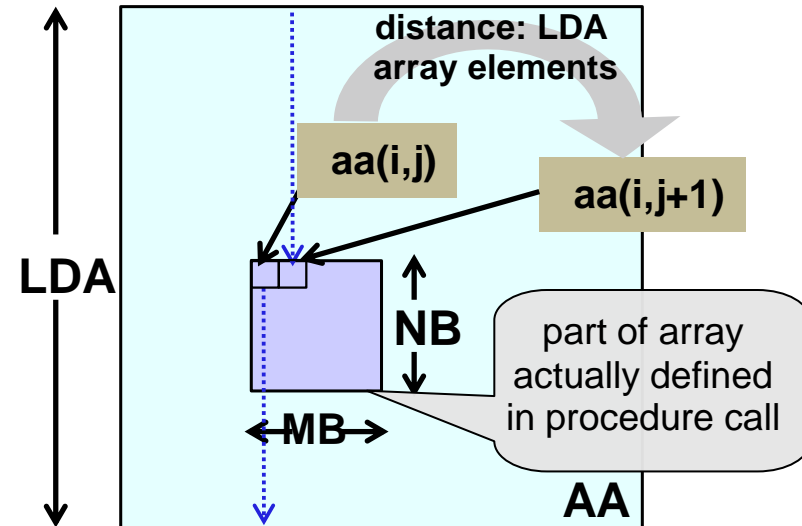
```
real :: aa(lda, lda)
:
: ! calculate i, j, nb, mb
call slvr(aa(i, j), lda, nb, mb)
```

pass **scalar** to procedure

- behaves as if address of specified array element is passed to procedure

Beware (for all usage patterns):

- avoid addressing outside storage area (e.g., MB too large for supplied array)
- „staircase effect“ if you get leading dimension wrong



aa(i, j) corresponds to **ad(1, 1)**
aa(i, j+1) corresponds to **ad(1, 2)**



■ Dummy array bounds

- declared via specification expressions

```
subroutine slvr_explicit( &
                        ad, lda, n, m)
  integer :: lda, n, m
  real :: ad( lda, n )
  ...
```

- also sometimes used in legacy interfaces

■ 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)

■ Example:

```
function add_real_int(r, i) &  
    result(ri)  
  
    real :: r(:)  
    integer :: i(:)  
    real :: ri(size(r))  
    integer :: k  
    do k = 1, size(r)  
        ri(k) = r(k) + real(i(k))  
    end do  
end function
```

■ Interface must be explicit

- shape of result evaluated at run time through use of a specification expression (at entry to function)

■ Usage

- conforming LHS required in an assignment

```
use ...  
implicit none  
integer, parameter :: ndim=...  
real :: r(ndim)  
integer :: ix(ndim)  
: ! initialize r, ix  
  
r = add_real_int(r, ix)
```

whatever module
the function is part of

■ 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

■ 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:

- 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 a non-elemental intrinsic
→ all elements must be evaluated in `log()`

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

■ 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)
```


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

■ 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**

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 preventing functional dependencies: checked by compiler.
For example: `cycle` or `exit` statements that exit the construct



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

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
-  has improvements (locality specifications, outside the scope of this course)



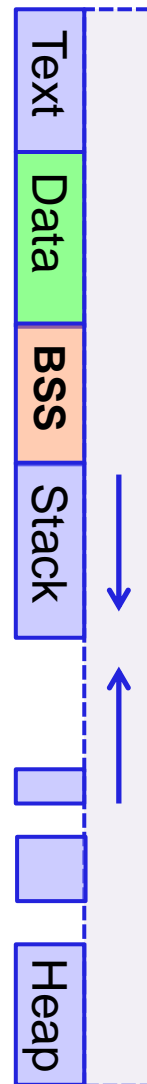
Dynamic Entities and Memory management

Some remarks about memory organization

Virtual memory

high address

- 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



low address

executable code (non-writable)

initialized global variables

static memory

uninitialized global variables („block started by symbol“)

Stack: dynamic data needed due to generation of new scope (grows/shrinks **automatically** as subprograms are invoked or completed; **size limitations** apply)

Heap: dynamically allocated memory (grows/shrinks under **explicit** programmer control, may cause **fragmentation**)

■ Defining all entities statically has consequences:

- need to check against defined size
- need to recompile often if size insufficient
- may not need large entities for complete duration of program run
- may run into physical memory limits (unlikely on systems with virtual memory if no default initialization is done)

■ Four mechanisms for dynamic provisioning of memory:

- ALLOCATABLE entities
- POINTER entities: can be, but need not be related to dynamic memory
- determine type as well as memory dynamically (data polymorphism, not treated in this course)
- automatic entities (already dealt with)



Beware:

- performance impact of allocation and deallocation
- fragmentation of memory

Allocatable objects (1)

Declaration

attribute

deferred shape → shape determined at run time
(**phi** is unallocated when execution starts)

```
real(dk), allocatable :: phi(:, :)
integer :: ifail, nd1, nd2
```

Allocation, use and deallocation

```
nd1 = ... ; nd2 = ...
allocate( phi(0:nd1, 0:nd2), stat=ifail )
if (ifail /= 0) stop 'procedure XXX: allocate failed'
do ...
    phi(i, j) = ...
end do
deallocate( phi )
```

contiguous memory area
for **phi** created on heap

phi is used for calculations ...

phi becomes deallocated
can usually be omitted
(auto-deallocation of non-saved objects)

- **It is optional and can be used in both allocation and deallocation**
 - a value of zero is returned if and only if the (de)allocation was successful
→ permits the programmer to deal with failures
 - without the `stat` argument, execution terminates if (de)allocation fails

Allocatable scalars

F03

deferred-length string

```
character( len=: ), allocatable :: dyn_string
type(body), allocatable :: a_body
:
allocate( character(len=64) :: dyn_string )
allocate( a_body )
:
do things with dyn_string and a_body
```

- this feature allows to determine size of character strings at run time (making the use of ISO_VARYING_STRINGS mostly obsolete)
→ **dynamic strings**
- otherwise relevant for polymorphic objects (not dealt with in this course) and parameterized derived types (see later)

■ Rationale: avoid undefined states and memory leakage

- an ALLOCATE statement must not be applied to an allocated object
- a DEALLOCATE statement must not be applied to an unallocated object

■ Supporting intrinsic

- the logical function ALLOCATED can be used in situations where the allocation status is not obvious
- example:

```
if ( allocated( phi ) ) then
    deallocate( phi )
end if
```

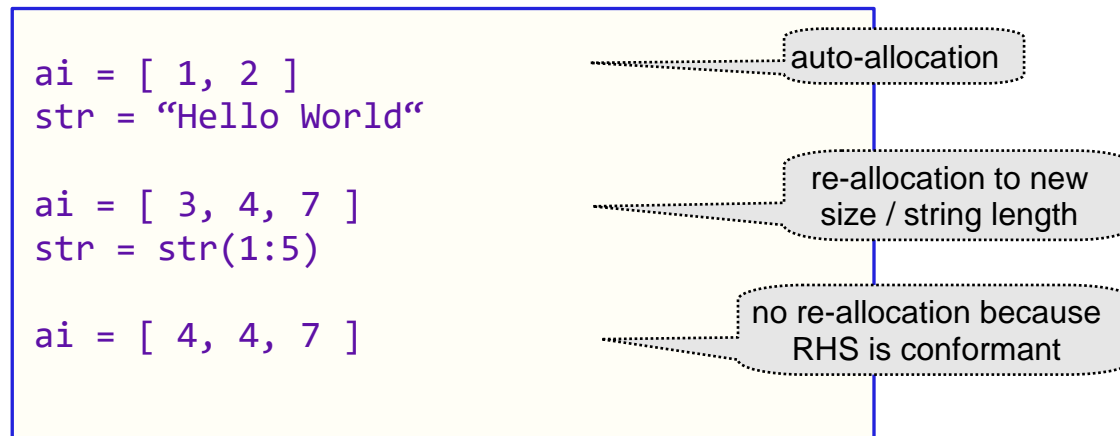
■ Allocatable variables with the (implicit or explicit) SAVE attribute

- allocation status is persistent (no auto-deallocation!)
- once allocated, object is persistent (until explicitly deallocated)

F03 Assignment to allocatable object

```
integer, allocatable :: ai(:)
character(len=:), allocatable :: str
```

- If LHS is unallocated or has wrong shape → auto-allocation to correct shape










■ **Note: this only works for assignment, not for an I/O transfer statement**

Apparent change of semantics?

Auto-allocation may be treacherous for legacy codes

- caused by vendor extensions that tolerate non-conforming array operations
- with new semantics may become conforming, yet deliver unexpected results

```
allocate(ai(2)) required F95  
ai = [ 1, 2 ]  
  
 ai = [ 3 ] non-conforming; may get [ 3, 2 ].  
  
 ai = [ 1, 3, 2 ] may get [ 1, 3 ].  
  
 ai(:) = [ 2, 3 ]
```

```
allocate(ai(2)) F03  
ai = [ 1, 2 ]   
  
ai = [ 3 ]  re-allocate to  
size 1, then size 3  
  
ai = [ 1, 3, 2 ]   
  
ai(:) = [ 2, 3 ]  non-conforming
```

- No reallocation happens with an array section LHS: shape conformance is programmer's responsibility
- compiler switches are usually available to revert to F95 behaviour, but it is better to fix your code

■ Intrinsic MOVE_ALLOC

```
call move_alloc(from, to)
```

- both arguments must have the ALLOCATABLE attribute
- to must be type and rank compatible with from

■ After execution:

- to has shape and value that from had at entry. If necessary, to is reallocated
- from is deallocated

■ Efficiency

- avoids an extra copy of data (basically, the descriptor is moved)

■ Usage example:

- efficient resizing of an array

```
real, allocatable :: &  
    x(:), auxil(:)  
integer :: new_size
```

assumption: x is allocated

```
new_size = ... ! larger than  
            ! size(x)  
allocate(auxil(new_size))  
auxil(1:size(x,1)) = x  
auxil(size(x,1)+1:) = ...  
  
call move_alloc(auxil, x)
```

- resizing might also involve shrinking, of course ...

Declaration:

deferred shape

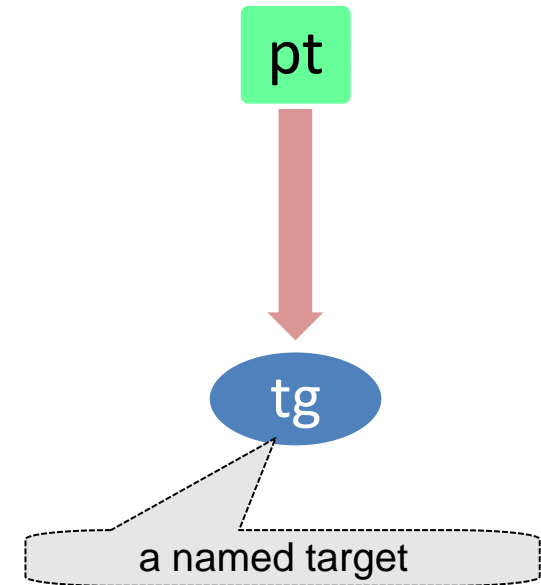
```
real(dk), pointer :: pt(:)  
real(dk), target :: tg(3)
```

- `pt` can be used as an **alias** for a real rank one array
- `tg` can be used as an object a pointer can be aliased against

Pointer assignment:

```
pt => tg
```

- causes `pt` to become associated (with `tg`)
- is a type/kind/rank-safe procedure (compile-time check of consistency)



Example:

```
real(dk), pointer :: pt(:)
real(dk), target :: tg(3)
```

```
pt => tg
```

```
pt(2) = 7.2
```

```
:
```

```
pt => null()
```

```
:
```

tg(2) is defined

now pt is disassociated

- pointer takes shape and bounds from target
- definitions and references to pointer operate on target

Symbolic representation

pt 

state after
declaration:
undefined

tg 

pt 

after pointer
assignment:
associated

tg 

pt 

assignment
to target via alias

tg 

pt 

disassociated after nullification

■ Creation of an **anonymous** TARGET

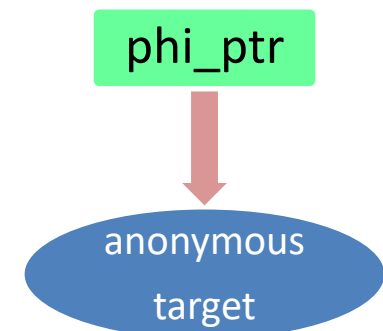
deferred shape → shape determined at run time

```
real(dk), pointer :: phi_ptr(:, :)  
integer :: ifail, nd1, nd2  
:  
  
nd1 = ... ; nd2 = ...  
allocate( phi_ptr(0:nd1, 0:nd2), stat=ifail )  
if (ifail /= 0) stop 'procedure XXX: allocate failed'  
:  
deallocate( phi_ptr )
```

1. **contiguous** memory area with implicit TARGET attribute created on heap
2. `phi_ptr` is pointer associated with it

`phi_ptr` is used for calculations

- use of DEALLOCATE is usually necessary for POINTER objects. Otherwise, memory leaks are likely to occur;
- the argument of DEALLOCATE must be a pointer to the **complete** anonymous target that was previously allocated;
- the ALLOCATED intrinsic **cannot** be applied to POINTER objects.



The ASSOCIATED intrinsic

■ A logical function that

- returns association status of an entity with POINTER attribute;
- it cannot be applied to an **undefined** POINTER

```
real(dk), pointer :: pt(:), qt(:)
real(dk), target :: tg(3)
```

```
pt => tg
write(*,*) associated(pt), associated(pt, tg)
```

prints T (.TRUE.), twice

```
allocate(qt(3))
write(*,*) associated(qt)
write(*,*) associated(pt, qt)
```

prints T (.TRUE.)

prints F (.FALSE.)

```
pt => null()
write(*,*) associated(pt)
```

prints F (.FALSE.)

Subobjects of a target

- also are targets

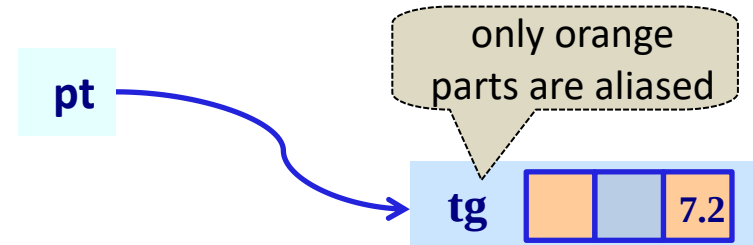
Example:

```
real(dk), pointer :: pt(:)
real(dk), target :: tg(3)
type(body), target :: bb(3)
:
pt => tg(1::2)
pt(2) = 7.2
:
pt => bb%mass
pt = 1.2
```

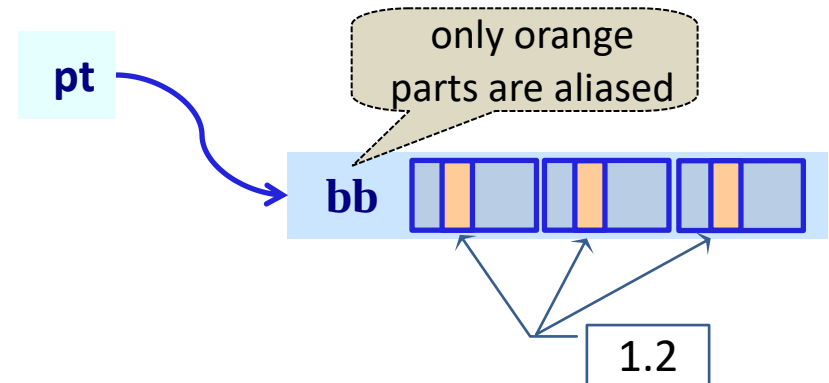
- pt associated with non-contiguous subobject

`is_contiguous(pt)` returns `.false.`

After first assignment:



After second assignment



■ Avoid the initially undefined state

- `null()` intrinsic function → start with disassociated state

```
real(dk), pointer :: pt(:) => null()
```

- F08 supports initialization with a non-allocatable TARGET (sub)object

```
real(dk), target, save :: x(ndim)  
real(dk), pointer :: pt(:) => x(:,2)
```

■ Initialization implies the **SAVE** attribute

- however for pointers it is only the association status that is preserved (because the values, if any, are stored in the targets)



Dangers using dynamically generated targets

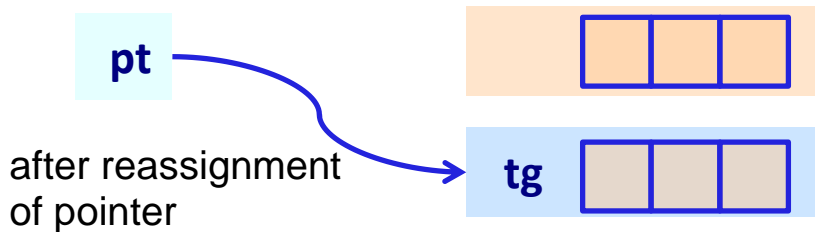
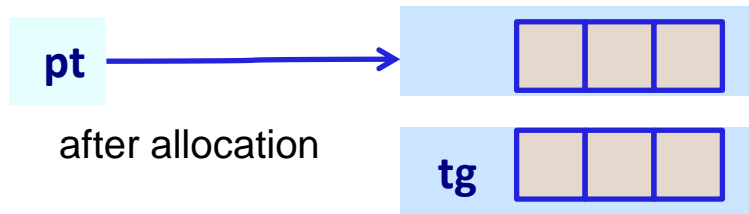
(Note: other methods for generating invalid pointers exist)

■ Potential memory leak

```

real, pointer :: pt(:) => null()
real, target :: tg(3)
allocate(pt(3))
pt => tg

```



- **unreachable** memory area created

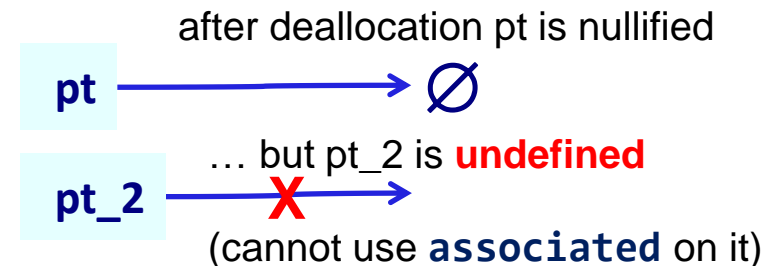
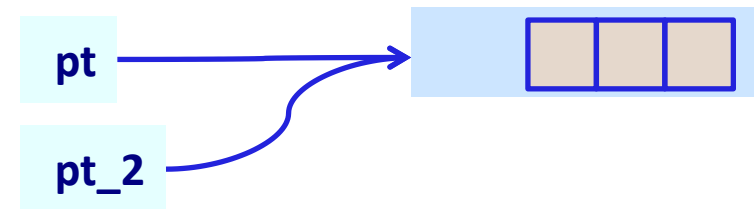
■ Undefined pointer after deallocation

```

real, pointer :: pt(:)=>null(),&
pt_2(:)=>null()
allocate(pt(3))
pt_2 => pt
deallocate(pt)

```

pt_2 has same target as pt

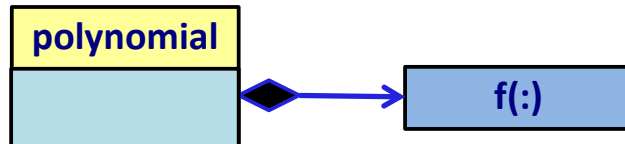


F03 Allocatable type components

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

default (initial) value is
not allocated

- a „**value**“ container

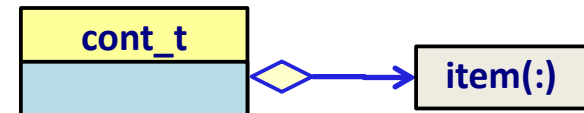


POINTER type components

```
type :: cont_t
  private
  real, pointer ::
    item(:) => null()
end type
```

default value is
disassociated

- a „**reference**“ container



Note: Container types will not be thoroughly treated in this course

■ Allocatable type components

```

type(polynomial) :: p1, p2
:
p2 = p1

```

define p1
(see e.g. next slide)

- assignment statement is equivalent to

```

if ( allocated(p2%f) ) &
    deallocate(p2%f)
allocate(p2%f(size(p1%f)))
p2%f(:) = p1%f

```

- „deep copy“

■ POINTER type components

```

type(cont_t) :: s1, s2
:
s2 = s1

```

define s1

- assignment statement is equivalent to

```

s2%item => s1%item

```

a reference,
not a copy

- „shallow copy“

Allocatable type components

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

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

When object becomes undefined

- allocatable components are automatically deallocated

usually will not happen for POINTER components

POINTER type components

```
type(cont_t) :: s1
real, target :: t1(ndim)
real, parameter :: t2(ndim) = ...
```

could be omitted (default initialized component)

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

- explicit target:

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

- not** permitted:

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

a constant cannot be a target

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

■ Irregularity:

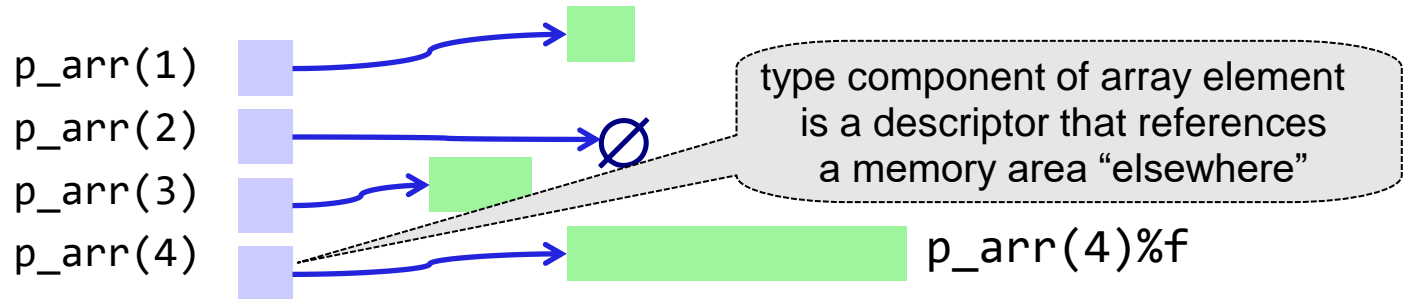
- each array element might have a component of different length
- or an array element might be unallocated (or disassociated)

```

type(polynomial) :: p_arr(4)

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

```



■ Applies for both allocatable and POINTER components

- a subobject designator like `p_arr(:)%f(2)` is **not** permitted

Allocatable and POINTER dummy arguments

(explicit interface required)

F03 Allocatable dummy argument

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

```
subroutine read_simulation_data(simulation_field, file_name)
  real, allocatable, intent(out) :: simulation_field(:, :, : )
  character(len=*), intent(in) :: file_name
  :
end subroutine read_simulation_data
```

deferred-shape

implementation allocates storage
after determining its size

POINTER dummy argument

- example: handling of a „reference container“

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

a private pointer type component

an exception to this will
be discussed in the
advanced course

Actual argument must have matching attribute

specified intent	allocatable dummy object	pointer dummy object
<code>in</code>	procedure must not modify argument or change its allocation status	procedure must not change association status of object
<code>out</code>	argument becomes deallocated on entry <div style="border: 1px dashed gray; border-radius: 10px; padding: 5px; width: fit-content; margin: 10px auto;">auto-deallocation of <code>simulation_field</code> on previous slide!</div>	pointer becomes undefined on entry
<code>inout</code>	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 are preserved across procedure invocations and pointer assignments

- Example:

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

What arrives inside `add_reference`?

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

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

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

F03 Explicit remapping of lower bounds is possible:

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

bounds are remapped for `a_container % item`

- A pointer of any rank may point at a rank-1 target
- Example:

```
real, allocatable, target :: storage(:)
real, pointer :: matrix(:, :), diagonal(:)
integer :: lb, ub, n
```

```
n = ... ; lb = ...; ub = lb + n - 1
allocate(storage(n*n))
```

```
matrix(lb:ub, lb:ub) => storage
diagonal => storage(:, n+1)
```

diagonal(i) now addresses
the same location as
matrix(lb+i-1, lb+i-1)

- requires specification of lower and upper bounds on LHS of pointer assignment

■ The CONTIGUOUS attribute can be specified for pointers

- (we already saw it for assumed-shape arrays)
- difference: **programmer** is responsible for guaranteeing the contiguity of the target in a pointer assignment

■ Examples:

- object `matrix` from previous slide

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

can be declared contiguous because whole allocated array storage is contiguous

- if contiguity of target is not known, check via intrinsic:

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

Allocatable function results (explicit interface required)

Scenario:

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

```
function deduplicate(x) result(r)
  integer, intent(in) :: x(:)
  integer, allocatable :: r(:)
  integer :: idr
  :
  allocate(r(idr))
  :
  do i = 1, idr
    r(i) = x(...)
  end do
end function deduplicate
```

find number idr of distinct values

Possible invocations:

- efficient (uses auto-allocation on assignment):

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

- less efficient (two function calls needed):

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

large enough?

- function result is auto-deallocated after completion of invocation


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

POINTER function results

(explicit interface required)

■ The POINTER attribute

- for a function result is permitted

 it is more difficult to handle on **both** the provider and the client side (need to avoid dangling pointers and potential memory leaks)

■ A reasonably safe example:


- extract section from container

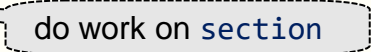
```
function get_section(c, s) result(r)
  type(cont_t), intent(in) :: c
  integer, intent(in) :: s(:)
  real, pointer :: r(:)
  r => null()
  if ( associated(c % item) ) &
    r => c % item(s(1):s(2):s(3))
end function get_section
```

checks on s omitted

- no anonymous target creation involved in this case!

- invocation:

```
type(cont_t) :: a_container
real, pointer :: section(:)
: 
section => get_section( &
  a_container, [start,end,stride] )

if ( associated(section) ) then
  : 
end if
```

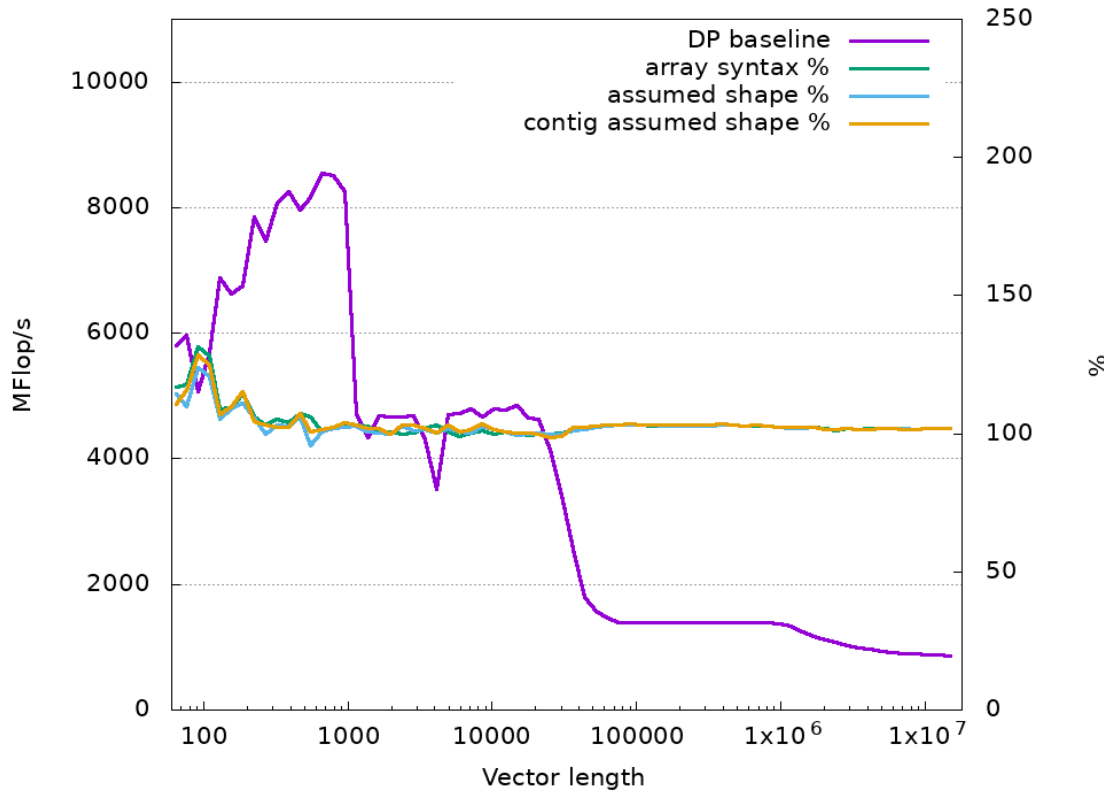
- note the **pointer assignment**
- it is essential for implementing correct semantics and sometimes also to avoid memory leaks

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



Further performance aspects and use of Parameterized derived types

Skylake 2.3 GHz with AVX512 / ifort 19.0



■ Array syntax

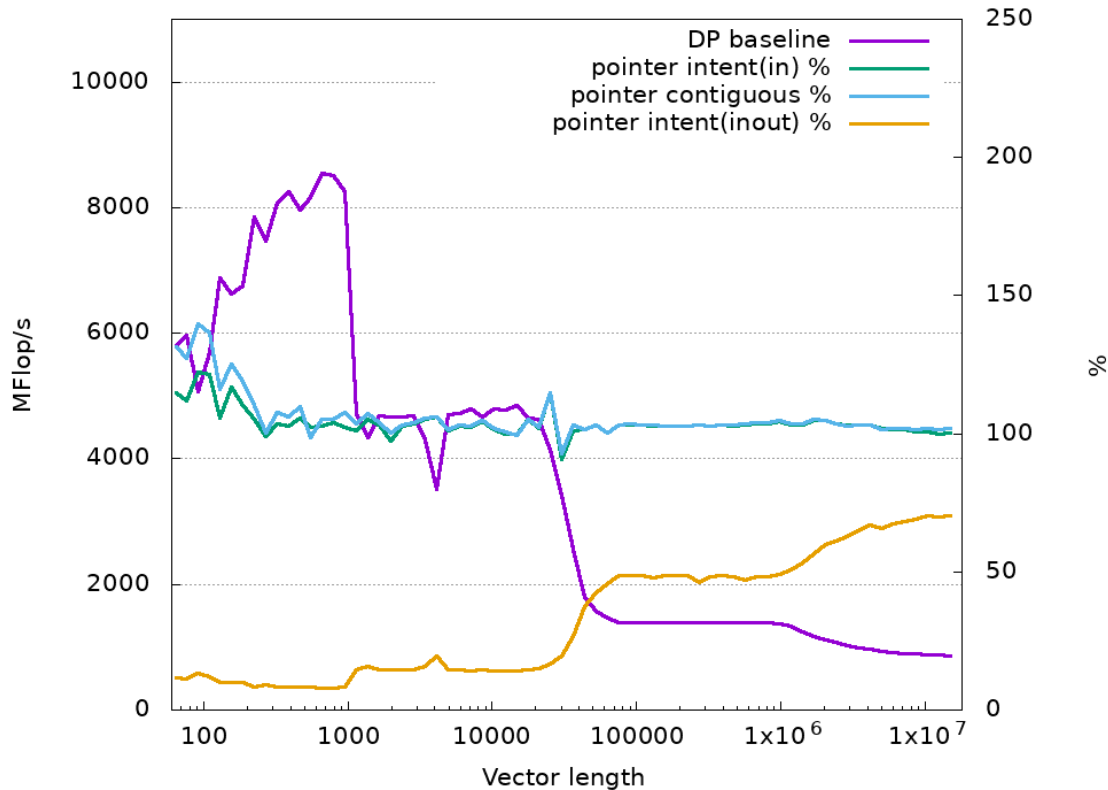
- $D(:) = A(:) + B(:)*C(:)$
fully optimized by compiler

■ Assumed shape array

- processing inside procedure
- in this simple case, compiler appears to generate multi-version code, so no difference to CONTIGUOUS case

POINTER dummy argument

Skylake 2.3 GHz with AVX512 / ifort 19.0

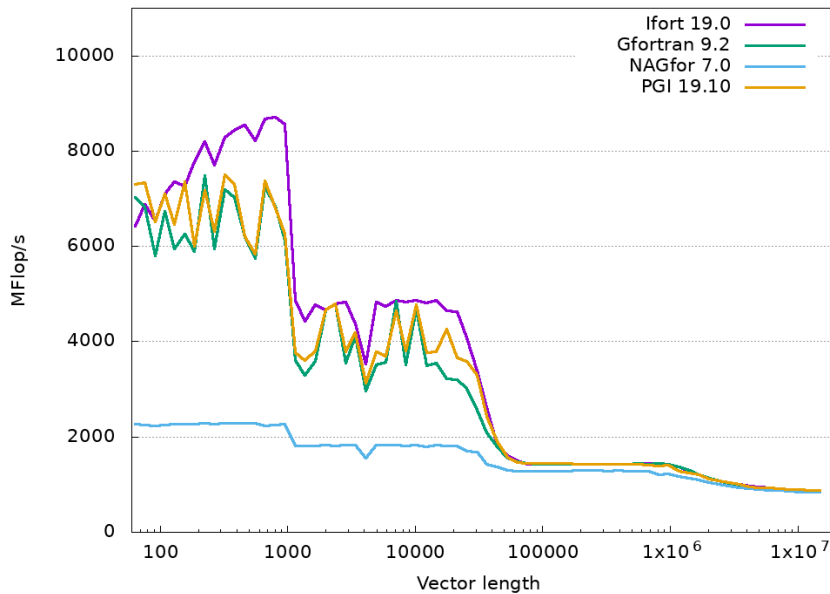


Actual aliasing is not happening

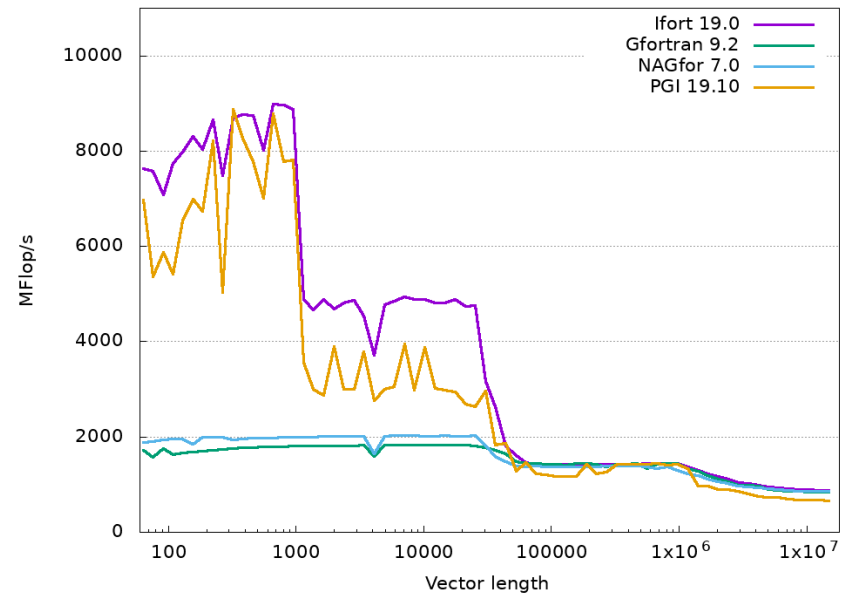
- Compiler sees this and performs vectorization for some cases anyway
- INTENT(inout) pointers perform quite badly

Comparing compilers (1)

Skylake 2.3 GHz Contiguous Assumed Shape



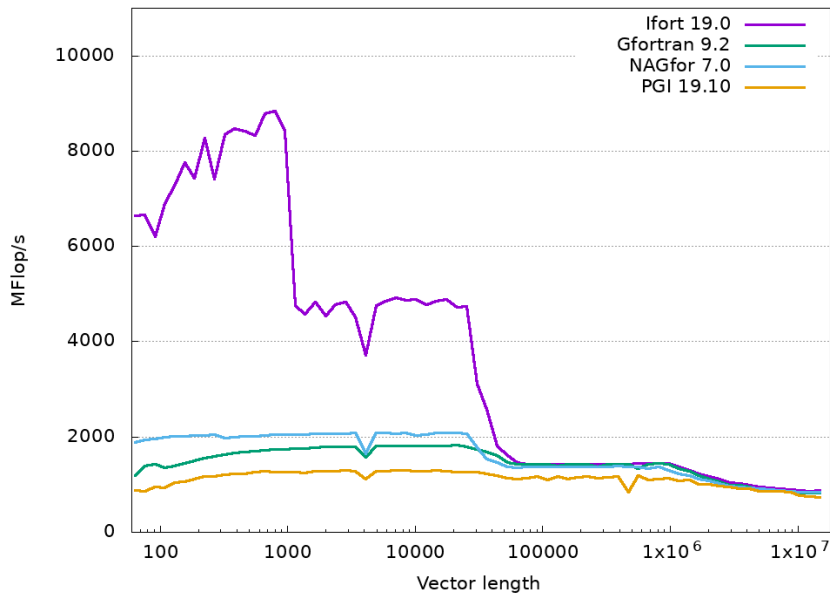
Skylake 2.3 GHz CONTIGUOUS POINTER



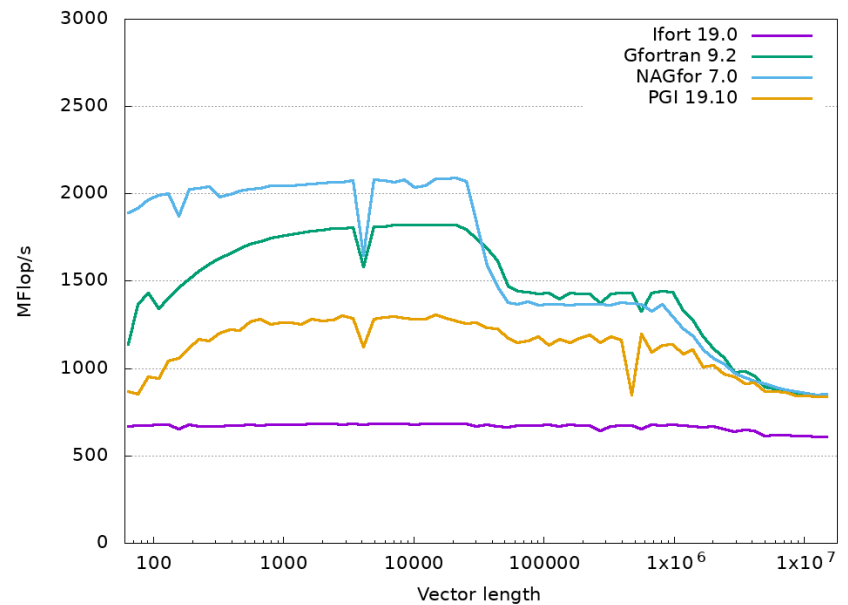
Comparing compilers (2)

Note the reduced maximum value

Skylake 2.3 GHz POINTER intent in



Skylake 2.3 GHz POINTER intent inout



Remember derived type (DT) "body":

```
module mod_body
  implicit none
  type :: body
    character(len=4) :: units
    real :: mass
    real :: pos(3), vel(3)
  end type body
contains
  subroutine kick(this, ...)
    ...
  end subroutine
end module
```

- "this" was declared a scalar

More typical usage:

- use an **array** to handle multiple bodies or a trajectory of a single body

```
use mod_body
:
type(body), allocatable :: traj(:)
:
allocate( traj(ntraj) )
call kick_s( traj, dp )
```

- requires a new variant of kick that handles **arrays of structure** (AoS)
- performance expectation?

Idea:

- fold array properties into type component → **structure of arrays** (SoA)
- this is achieved via integer-typed parameters, which become part of the type

Two variants: KIND and LEN (length) parametrization

- semantic difference: compile-time vs. run-time resolution of parameter values

```
type :: body_p( k, ntraj )
  integer, kind :: k = kind(1.0)
  integer, len :: ntraj = 1
  character(len=4) :: units
  real(kind=k) :: mass(ntraj)
  real(kind=k) :: pos(ntraj,3), vel(ntraj,3)
end type body_p
```

reuse for different representations

default values are permitted

reduce overhead

array dimension folded into component

■ Static declarations

- **unspecified** type parameters take default values; specification is **obligatory** if no defaults exist

```
type( body_p (ntraj=ndim) ) :: traj_ndim
type( body_p ) :: traj_1
type( body_p (k=kind(1.0d0), ntraj=ndim) ) :: dptraj_ndim
```

constant expression required

■ Dynamic objects

- length type parameters are usually **deferred**:

```
type( body_p (ntraj=:) ), allocatable :: dyn_traj
```

a PDT scalar

(might also be an allocatable dummy argument)

- allocation requires a **type specification**:

```
allocate( body_p (ntraj=12) :: dyn_traj )
write(*,*) 'Shape of vel component: ', shape( dyn_traj%vel )
```

value is [12 , 3]

Type parameter inquiry

- **Type parameters are also type components**
 - special case: **read-only** access
 - Example:

```
write(*,*) dyn_traj % k, dyn_traj % ntraj
```

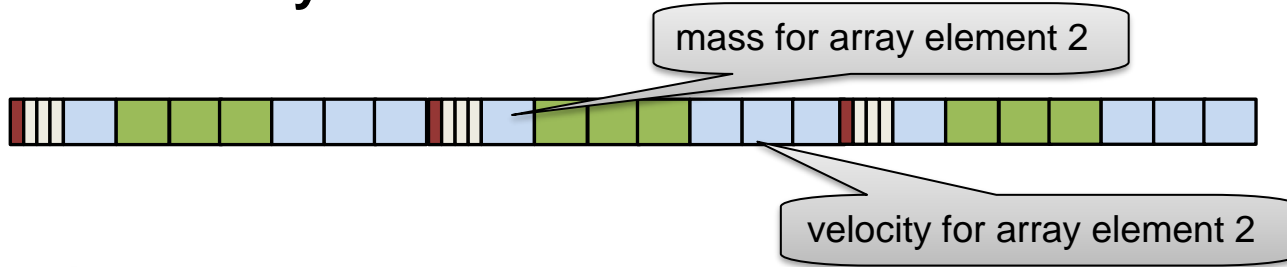
produces the output

```
4 12
```

(4 is the KIND number used for default real by Intel Fortran)

Comparing AoS vs SoA Memory Layout

■ AoS with 3 array elements



■ SoA with LEN parameter value 3



actual layout may differ in details

■ Memory area colored blue is referenced or defined by "kick"

- AoS has effective stride, especially for „mass“ component → loss of spatial locality, independent of array size
- AoS vectorization length is 1 and 3, respectively
- SoA always uses contiguous memory for both components
- SoA can be fully vectorized for sufficiently large fields

■ Requires special syntax for dummy argument declaration

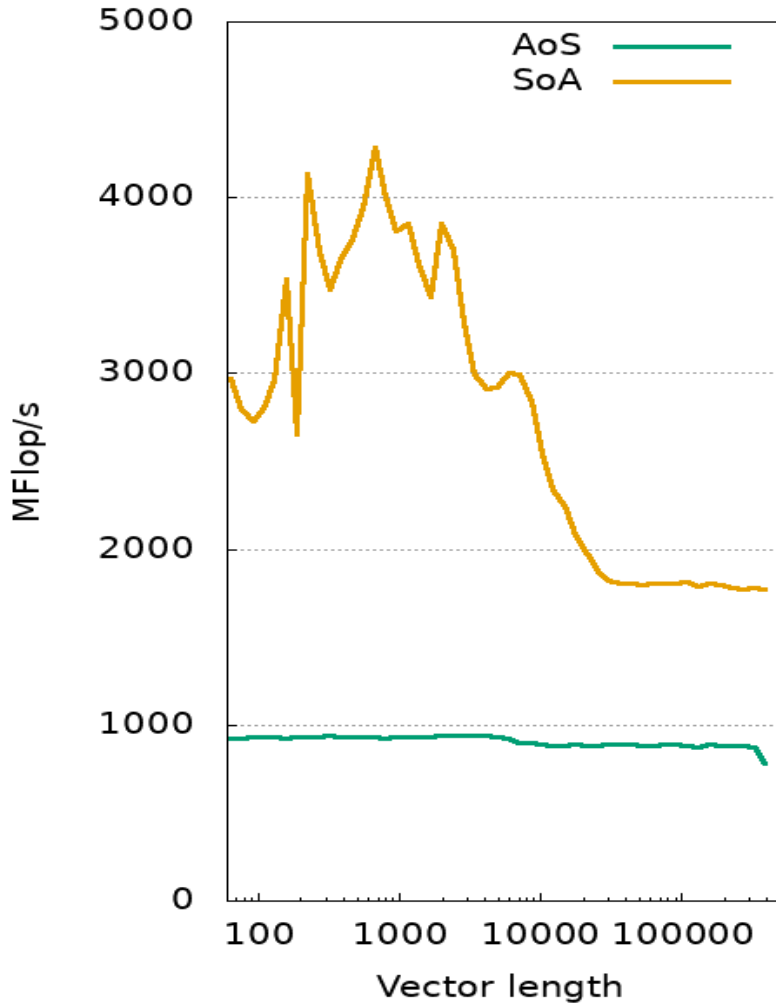
```
subroutine kick_p( bowling_ball, dp )  
  
    type(body_p( k=kind(1.0), ntraj=* )), &  
                                intent(inout) :: bowling_ball  
    real, intent(in) :: dp(:, :)  
    :  
end subroutine kick_p
```

- KIND type parameter requires compile-time constant as specification. Each value requires its separate procedure
- LEN type parameter is declared as being **assumed** from the actual argument

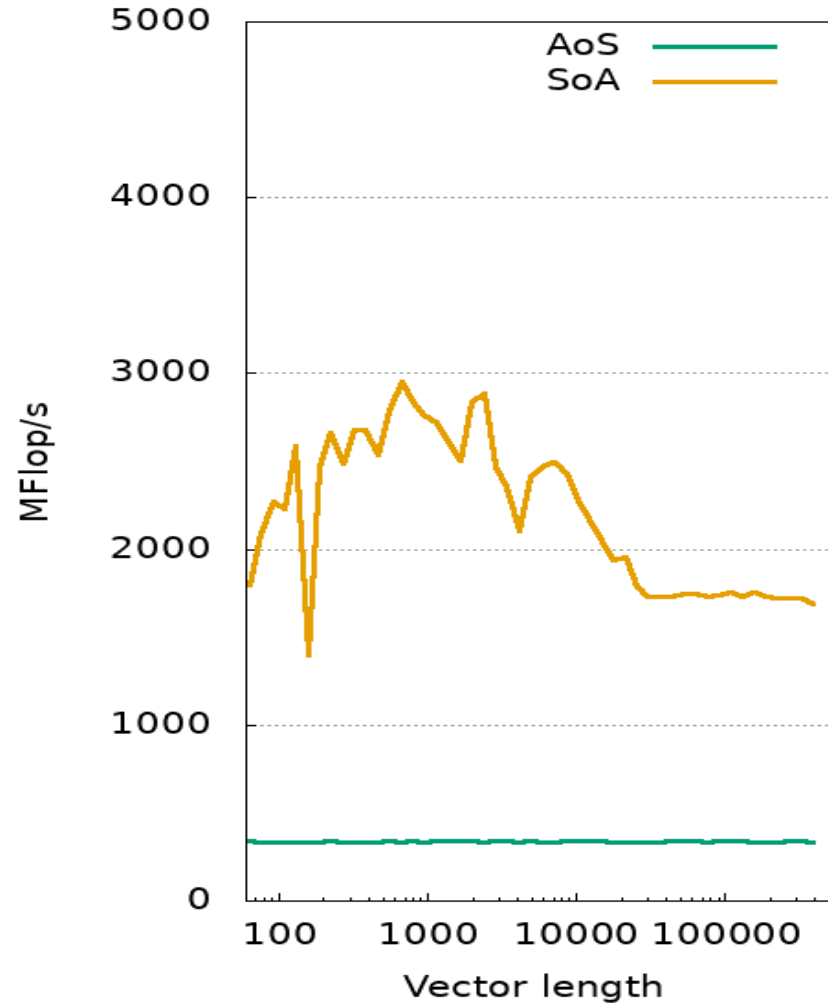
Performance Comparison on an Intel Skylake 2.3 GHz base frequency core



ifort 19.0



gfortran 9.2



■ Only part of the PDT semantics was covered here

- come to the "Advanced Fortran Topics" course for more

■ PDTs were one of the latest-implemented **F03** features

- compilers are still rather buggy in places, but simple scenarios such as shown here should now work

■ Container types permit similar optimizations

- vectorization should work
- however, irregular memory layout can cause difficulties

e.g., allocatable
type components

■ More detailed control of vectorization

- might be achieved by using OpenMP SIMD directives
- check compiler vectorization report!



Generic interfaces and overloading

Generic Interfaces (1)

Basic idea

- invoke procedures that „do the same thing“ for differently typed arguments by the **same name**

Precedent: intrinsics already work that way.

For example, `sqrt` will work for real arguments of any kind, as well as for complex arguments

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

- both default and high precision versions of `wsqrt` should be usable by the same name
- achieved by specifying a **named interface** that lists the **specific** procedures

```

module mod_functions
  interface wsqrt
    procedure wsqrt
    procedure wsqrt_dk
  end interface wsqrt
  private :: wsqrt_dk
contains
  real function wsqrt(x, p)
    real, intent(in) :: x, p
    ...
  end function wsqrt
  real(dk) function wsqrt_dk(x, p)
    real(dk), intent(in) :: x, p
    ...
  end function wsqrt_dk
end module

```

also permitted: **module procedure**

only expose the generic name

dk specifies non-default real

Rules:

- specifics must be either **all** functions or **all** subroutines
- external procedures also possible
- one specific **per module** may itself have the generic name

■ Invocation

```
use mod_functions
implicit none
real :: x,p
real(dk) :: xd, pd
```

initialize variables

```
:
```

```
write(*,*) wsqrt(x,p)
```

invokes specific `wsqrt`

```
write(*,*) wsqrt(xd,pd)
```

invokes specific `wsqrt_dk`

```
write(*,*) wsqrt(x,pd)
```

no matching specific exists → **rejected** by compiler

■ Distinguishability:

(only the most relevant rules listed here)

- at least one non-optional argument must be different with respect to either type, kind or rank (**TKR**),
- or differ by being a dummy procedure argument as opposed to a dummy data argument

F08

■ Specific functions

- must have sufficiently different interface
- invocations always determined at **compile** time

■ The following generic

(which legitimately references interfaces of external procedures)

```
interface foo
  subroutine foo_1(i, r)
    integer :: i
    real :: r
  end subroutine
  subroutine foo_2(r, i)
    integer :: i
    real :: r
  end subroutine
end interface foo
```

is **non-conforming**, since the call

```
integer :: j
call foo(i=j, r=2.0)
```

cannot be unambiguously resolved.

■ TKR rule

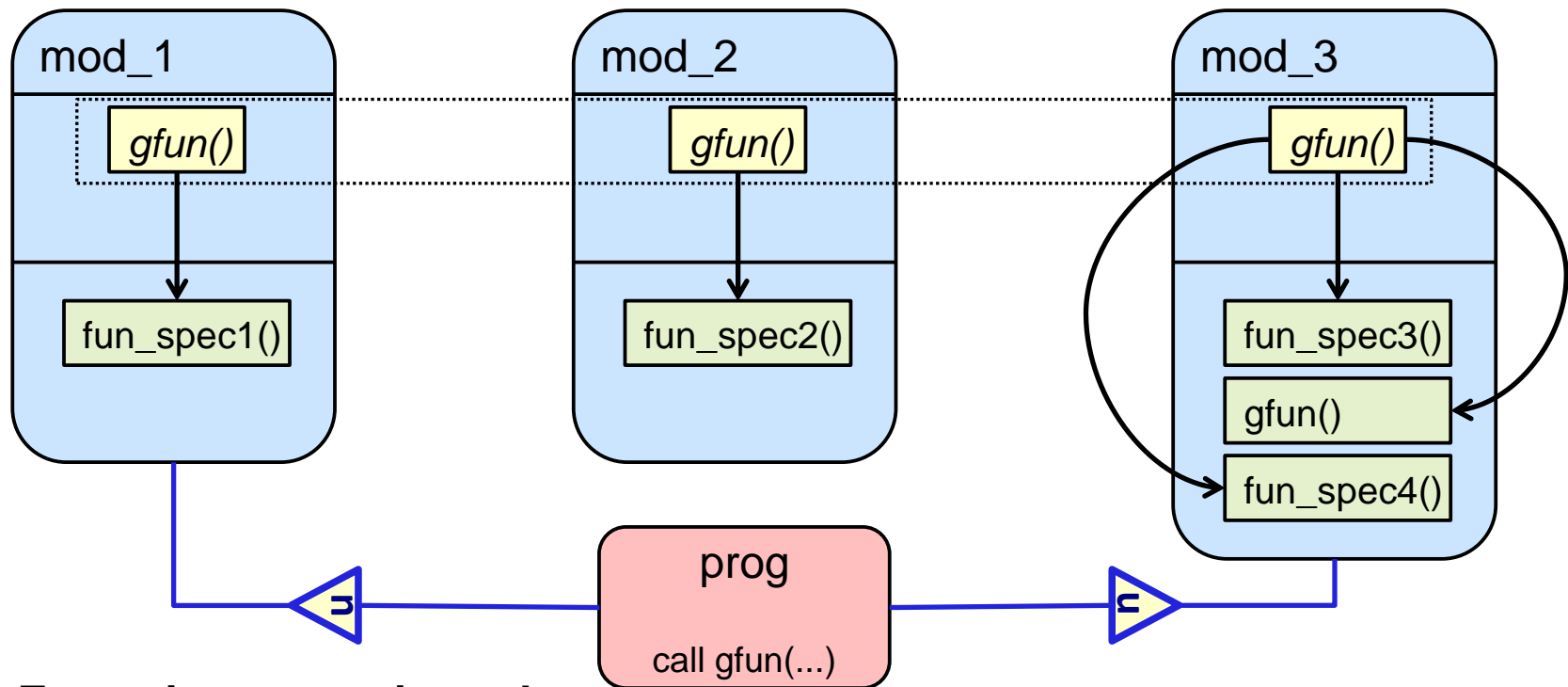
- is easy if numbers of non-optional arguments differ
- may need to also account for permutations of arguments if not

■ When does it **not** make sense to use a generic?

- to get around name space problems → using encapsulation (**only** clause) or renaming are better alternatives in this case
- danger of functional confusion (code using the generics becomes difficult to read)

Generic Interfaces (4)

Working across module boundaries



■ Exception to naming rules

- with generics, same name can be re-used in different modules

■ Unambiguous resolution:

- also depends on which specifics are accessed
- *gfun()*: interface of *fun_spec2()* might be ambiguous with respect to other specifics (not recommended!), since not use associated by „prog“

Generic Interfaces (5)

Arrays of differing rank

- Write a generic that supports an actual argument of multiple ranks
- Assumed shape dummy argument
 - somewhat troublesome – may need to write 15 specific interfaces for every argument to cover all possible ranks (16 if scalars are included)
- Assumed size dummy argument
 - when defining generic interfaces with such an argument, a rank mismatch between actual and dummy argument is **not** allowed
 - this is different from using a specific call – but in the latter, scalar arguments cannot participate
 - and the argument size typically must be specified as a separate argument
- A new feature in **F18** is available ...



■ Scenario:

- An algorithm is considered that can handle problems of different dimensionality
- The functionality cannot be handled by an ELEMENTAL procedure

■ Consequences:

- the computational interface should supply a single specific that can handle calls with arrays of arbitrary rank
- it should be also possible to use this as a specific in a generic interface (e.g., because types might also be varied)

```
use mod_io
:
real :: a, b(nbdim), c(ncdim1,ncdim2), d(nddim1,nddim2,nddim3)
: ! supply all values

call write_iobuf(a, outfile_a)
call write_iobuf(b, outfile_b)
call write_iobuf(c, outfile_c)
call write_iobuf(d, outfile_d)
```

Assumed-rank dummy argument (2) and SELECT RANK block construct

F18

Declaration

- requires explicit interface

```
module mod_io
:
contains
  subroutine write_iobuf(buf,file)
    real, intent(in) :: buf(..)
    character(len=*), &
      intent(in) :: file
    : ! open file
    :
  end subroutine write_iobuf
end module mod_io
```

- also permitted: explicit DIMENSION(..) attribute
- no references or definitions are possible, except certain array inquiries (e.g. RANK(), SHAPE())

Run time rank resolution

- a new block construct

```
select rank (buf)
  rank (0) write(iu) buf
  rank (1) write(iu) buf(:)
  rank (2) write(iu) buf(:, :)
  rank (3) write(iu) buf(:, :, :)
  rank (*) stop 'assumed size unsupported'
  rank default stop 'rank > 3 unsupported'
end select
```

actual argument is scalar

actual argument is assumed-size

- inside each block, object is of designated rank, and references and definitions are permitted
- at most one block gets executed

Named interface with same name as a derived type

- has the same accessibility as the type (as possibly opposed to its components)

```

module mod_date
  : ! previous type definition for date
  interface date
    module procedure create_date
    module procedure create_date_safe
  end interface
contains
  type(date) function create_date(day, mon, year)
    integer, intent(in) :: day, mon
    integer, intent(in) :: year
    : ! check constraints on input
    create_date%day = day; ... ! define object
  end function
  type(date) function create_date_safe(day, mon, year)
    integer, intent(in) :: day
    character(len=3), intent(in) :: mon
    integer, intent(in) :: year
    : ! implementation omitted
  end function
end module mod_date

```

any number of
specific functions

must be a function with scalar result

provide additional
semantics

obliged to use
component notation

improve safety of use
via a suitably chosen
interface signature

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

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

■ Example from previous slide continued:

```
use mod_date  
type(date) :: o_d1, o_d2
```

```
o_d1 = date(12, 10, 2012)
```

invokes `create_date`
(same syntax as
structure constructor)

```
o_d2 = date(day=12, &  
           mon='Oct', &  
           year=2012)
```

invokes `create_date_safe`

■ Implement additional semantics not available through structure constructor e.g.,

- enforce constraints on values of type components
- provide a safe-to-use interface
- handle dynamic type components (see later)

■ Type for rational numbers (also an exercise)

```
module rational
  implicit none
  type :: fraction
    :
    :
  end type
  :
end module
```

type components
etc. omitted

- **For fractions, operations like +, -, *, / exist, mathematically**
 - but these will not „simply“ work for the above-defined derived type
- **Fortran permits defining extensions of these for derived types**
 - both numeric and non-numeric (e.g. //, .or.) operators can be extended

■ Example: add fractions

```
module rational
:
interface operator(+)
  module procedure add_fi
  module procedure add_fl
end interface
contains
function add_fi(f1, f2) result(r)
  type(fraction), &
  intent(in) :: f1, f2
  type(fraction) :: r
  :
end function
function add_fl(f1, f2) result(r)
:
end function
end module
```

previous type definition

exactly two dummy arguments

same for a different type `fraction_1` that uses „long“ integers

- restricted named interface

■ Usage:

```
use rational
type(fraction) :: x, y, z
:
x = y + z
```

define y, z

invokes `x = add_fi((y),(z))`

■ Further rules:

- both dummy arguments must be `intent(in)`
- for a **unary** operator, a single dummy argument with `intent(in)` must be specified
- existing intrinsic operators **cannot** be changed

Programmer-defined operators

■ Example: convolution

$$f_i = \sum_{j \leq i} op_{i-j+1} \cdot vc_j$$

- a (binary) operation not covered by an intrinsic operation

```

module user_ops
  interface operator (.convolve.)
    module procedure conv
  end interface
contains
  function conv(op, vc) result(r)
    real, intent(in) :: op(:),vc(:)
    real :: r( size(vc) )
    :
  end function
end module

```

implementation not shown here

■ Usage:

```

use user_ops
integer, parameter :: ndim=100
real :: x(ndim), op(ndim)

```

```

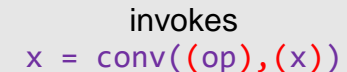
: 

```

```

x = op .convolve. x

```

 invokes
x = conv((op),(x))

■ Further rules:

- generic name can have up to 31 characters between dots
- otherwise same rules as for intrinsic operations

■ Overloaded intrinsic operators

- obey the **same** precedence rules than their intrinsic counterparts
- usual left-to-right evaluation (except for **)

■ Semantic aspects:

- for (different) derived types, the overloading should obey associativity
- possible performance issue (A derived type, B and C intrinsic type):

```
X = (A*B)*C; Y = A*(B*C)
```

- both expressions are valid, but the second one is typically faster

- parentheses for readability and correctness if multiple operators are overloaded
- example: for A, B, and C of derived type, with overloaded + and *

```
X = A + B * C
```

is by default evaluated as

```
X = A + (B * C)
```

■ Unary defined operators

- have **higher** precedence than any other operator

■ Binary defined operators

- have **lower** precedence than any other operator

■ Parentheses may be vital

```
X = A .convolve. B + C
```

is evaluated as

```
X = A .convolve. (B + C)
```

which very probably is not what you meant.

- what you meant must be written

```
X = (A .convolve. B) + C
```

■ Renaming of defined operators on the USE line F03

```
use user_ops, operator(.conv.) => operator(.convolve.)
```

- however, this is not allowed for intrinsic operators

■ Generic resolution against elemental specifics

- if both an elemental and a non-elemental specific match, the non-elemental specific is used

■ Overloading intrinsic procedures

- is allowed, but will render the intrinsic procedure **inaccessible** if it has the same interface
- is definitely not recommended unless interface is sufficiently different

■ Generic names **cannot** be used as procedure arguments

- for generic intrinsics, there exists a whitelist

- **Default assignment is unavailable between objects of different derived types**
- **Default assignment for derived types might not have the desired semantics**
 - especially for container types (see advanced course)

This motivates a desire for overloading the assignment ...

■ Uses a restricted named interface:

```
module rational
  : ! type definition
  interface assignment(=)
    procedure assign_from_int
  end interface
contains
  subroutine assign_from_int(r, x)
    type(rational), intent(out) :: r
    integer, intent(in) :: x
    :
  end subroutine
end module
```

exactly two arguments

- here, a conversion is implemented

■ Further rules:

- first argument must be `intent(out)` or `intent(inout)`
- second argument must be `intent(in)`
- assignment **cannot** be overloaded for intrinsic types (as both first and second arguments)
- overload usually wins out vs. intrinsic assignment (if the latter exists)
Exception: implicitly assigned aggregating type's components → aggregating type must also overload the assignment

Now we proceed to an **exercise session**



Input and Output to external storage

■ (logical) Record:

- sequence of values or characters

■ Types of records:

- **formatted**: conversion between internal representation of data and external form
- **unformatted**: same representation of internal and external form
- **endfile**: last record of a file; may be implicitly or explicitly written
- external form: operating environment dependency

■ File:

- sequence of records
- records must be all formatted or all unformatted

■ Types of files:

(nearly independent of record type)

- **external**: exists on a medium outside the program
 access methods: sequential, direct-access and stream access
- **internal**: memory area which behaves like a file (used for conversion between data representations)

■ File operation I/O statements

- manage connection of external files with the program
- determine mode or kind of I/O
- most important statements: **OPEN, CLOSE, INQUIRE**
- navigate inside file: **BACKSPACE, REWIND**

■ Data transfer I/O statements

- read, generate or modify records inside files
- most important statements: **READ, WRITE**

■ Arguments for data transfers:

- objects to be transferred: **I/O list**
- transfer method: **I/O control specification**
- specifically for formatted records: **I/O editing** – an important part of the control specification

Concept of I/O unit

■ Abstraction:

- allows the program to refer to a file
- via a default integer,
- which is part of the **global state** of the program

■ Pre-connected units:

- units associated with a (special) file **without** executing an OPEN statement
- special notation: star instead of integer
- standard output `write(*, ...) ...`
- standard input `read(*, ...) ...`
- error unit: this is where error messages from the run time library typically are written to. May be the same as standard output

■ Alternative:

- replace star by constants defined in ISO_FORTRAN_ENV:

```
use, intrinsic :: iso_fortran_env
write(output_unit, ...) ...
read(input_unit, ...) ...
```

or to **error_unit**

Associating a file with a unit – The OPEN and CLOSE statements

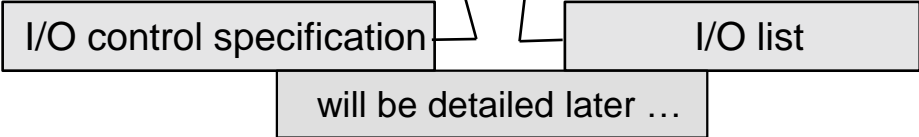
■ Example:

- opening a (sequential) formatted file for reading only

```
integer :: iunit
: ! define iunit
open(unit=iunit, &
      action='READ', &
      file='my_file', &
      form='FORMATTED', &
      status='OLD')
read(iunit, ...) ..
```

the keyword „unit=“ is optional

a write statement is not permissible here



■ A unit may only be associated with **one file at a time**

- and vice versa
- close the file to disassociate

```
! ... continued from before
close(unit=iunit)

open(iunit, &
      action='WRITE',
      file='new_file', &
      form='FORMATTED')
write(iunit, ...) ..
```

need not be in same program unit as close

a read statement is not permissible here

Identifying a usable I/O unit

■ A given unit number

- need not exist (some may be reserved)
- may already be in use
- perform **inquiry by unit**:

```
logical :: unit_exists, &
        unit_used
iunit = ...
inquire(iunit, &
        exist=unit_exists, &
        opened=unit_used)
```

some non-negative integer

if unit_exists is set to **.true.** and unit_used is set to **.false.**, an open on iunit will succeed.

■ Note:

- Shell/OS limit on number of filehandles – not a Fortran issue

F08

Improved method:

- use the **newunit** specifier in open:

```
integer :: iunit
:
open(newunit=iunit, ..., &
     file='myfile', ... )
```

- this will **define** iunit with a (negative) integer that is connected to the specified file.

■ I/O list:

- list containing all objects for which I/O is to be performed
- may include an **implied-DO** list, otherwise comma-separated items
- **read:** variables
- **write:** variables or expressions (including function calls)

■ Array items:

- I/O in array element order:

```
write(iu, *) a(1:3)
write(iu, *) a(1),a(2),a(3)
write(iu, *) (a(i),i=1,3)
```

implied-DO loop

the three statements are equivalent

- an array element may not appear more than once in an input statement

■ Derived type objects

- transfer in order of type components for POD types
- „container types“ require UDDTIO

■ Dynamic entities

- must be allocated/associated
- for pointer variables, the target is transferred

■ Empty I/O list

- no object specified, or
 - zero-trip implied-DO
- writes an empty record, or shifts file position to next record upon `read()`

List-directed I/O

■ A statement of the form

```
write(iunit, fmt=*) a, b, c
```

keyword "fmt=" is optional. It stands for the word "format".

- writes all items from the I/O list to the unit
- in a **processor-dependent** format (including record length)

■ Resulting file can be (portably) processed with **list-directed input**

```
integer :: iunit, i
real :: z(7)
character(len=20) :: c
logical :: w
:
read(iunit, fmt=*) i, &
      z(1:6), c, w
```

open the unit
on `my_file`

reading „z“ would fail

- **Note:** slash in input field terminates I/O statement.

contents of `my_file`

```
5 .3 , , -1.2E-1, 1.4
2*3.6 "No guarantee" .T.
```

blanks and/or comma
are value separators

quotes prevent unwanted splitting

variable `z(1:6)` after I/O statement

```
.3 U -.12 1.4 3.6 3.6
```

null value in input field

repeat factor

■ Give programmer means

- to permit specification on how to perform **formatted** I/O transfer
- via a parenthesized character expression - a **format string**

■ This uniquely defines

- conversion from character string representing an I/O record to internal representation (or vice versa)

■ Three classes of edit descriptors:

- **data** edit descriptors (associated with the way an I/O item of a specific type is converted)
- **control** edit descriptors (refers to the specific way a record is transferred)
- **character** string descriptor (embed a string in the character expression → usually used for output)

- **Embed a string in a format specification**

- applies only for **output**

- **Example:**

```
write(*, fmt='(i5, '' comes before ',i5)') 22, 23
```

will produce the character sequence

```
bbb22bcomesbbeforebbbb23
```

- **Note:** repeated single quote masks a single one inside format string

Table of data edit descriptors

Descriptor	type of list item	specific function
A	character	
B	integer	conversion to/from binary
I	integer	
O	integer	conversion to/from octal
Z	integer	conversion to/from hexadecimal
D	real	indicate extended precision and exponent
E	real	indicate exponent
EN	real	engineering notation
ES	real	scientific notation
EX	real	hexadecimal notation
F	real	fixed point (mostly ...)
L	logical	
G	any intrinsic	general editing: „auto-detection“ of edit descriptor to use
DT	derived type	user-defined „object-oriented“ I/O (aka UDDTIO)

also used for
complex types

Items marked green will be explicitly mentioned

Table of control edit descriptors

Descriptor(s)	function	comments
BN, BZ	handling of embedded blanks in input fields	ignore / insert zero
SS, SP, S	output of leading signs	suppress/enforce/processor-defined
<i>kP</i>	scale numbers on input (or output)	usually by factor 10^{-k} (or 10^k), except for scientific representation
<i>Tn, (TRn nX), TLn</i>	tabulation inside a record	move to position / right / left (<i>n</i> in units of characters)
<i>/</i>	generate a new record	„linefeed“
<i>:</i> colon	terminate format control	when running out of I/O items
RU, RD, RZ, RN, RC, RP	change rounding mode for connection	up, down, to zero, to nearest, compatible, processor-defined

Items marked green will be explicitly mentioned

■ Format argument may be

- an asterisk → list-directed input or output as previously discussed,
- a default character expression specifying an explicit format, or
- a statement label referencing a (non-executable) **format** statement

■ Examples:

```
character(len=10) :: my_fmt
real :: r
:
my_fmt = '(e10.3)'
r = 2.33444e+2
open(iu, ...)
write(iu, fmt=my_fmt) r
write(iu, '(e11.4)') r
write(iu, fmt=1001) r
:
1001 format(e12.5)
end
```

Recommended
method

Note: format variable may **not**
be part of I/O **input** list

internal I/O (see later)
allows to **dynamically** define format

output might be:

```
b0.233E+03
b0.2334E+03
b0.23344E+03
```

blanks indicated by "b"

If you use labeled formats, collect them
near the end of the subprogram, with
number range separate from other labels

Using data and control edit descriptors (1)

```
real :: x(2); integer :: i(3)
character(len=3) :: s = 'def'
x = [2.331e+1, -.7151]; i = [7,9,-3]
```

Field width and repeat factor

```
write(iu, '(2E10.3,3I2,A2)') x, i, s
```

repeat count

Output will be

```
b0.233E+02-0.715E+00b7b9-3de
```

blanks indicated by "b"

field width is 10 (includes sign)

width 2 – blank padding if not all characters needed

Bracketing and tabulation

```
write(iu, '(2(F5.2,1X,I2))') x(1),i(1),x(2),i(2)
```

repeat count applies to parenthesised expression

Output will be

```
23.30bb7-0.72bb9
```

control edit descriptor for right tabulation
inserts a single blank

```
integer, allocatable :: csv_list(:)
allocate(csv_list(5))
csv_list(:) = [ 1, 2, 3, 4, 5 ]
```

■ Unlimited repeat count and colon editing

```
write(iu, '(*(I2,:,',''))') csv_list
```

only permitted on last item of format string

terminates output if data items run out

Output for above value of `csv_list`
`b1,b2,b3,b4,b5`

no comma at the end

■ Force record split

```
write(iu, '(*(3I2,/))') csv_list
```

Output:
`b1b2b3`
`b4b5`


```
integer :: i  
real(dk) :: x
```

■ Format overflow on output

```
i = 12345  
write(iu, '(i3)') i  
x = -1.532E102  
write(iu, '(e8.4)') x  
write(iu, '(e10.4)') x  
x = 1.6732E7  
write(iu, '(f7.1)') x
```

Output File contains:

```
***  
*****  
-.1532+103  
*****
```

■ Input variables **undefined**

Input File contains:
12345
-1.532E+102
*b*1.6732E+07

```
read(iu, '(i9)') i  
read(iu, '(e8.3)') x  
read(iu, '(e18.4)') x
```

- due to inconsistent width
(note that number of decimals is usually ignored on input)
- RTL might terminate program

... and how to avoid them

■ On output

- width \geq digits + 7 for scientific notation
- specify exponent width for sc. not.
- width(number, digits) for fixed point
- width(number) for integer

■ Alternative

- automatic width adjustment for fixed point or integer

```
i = 12345
write(iu,'(i0)') i
x = -1.532E102
write(iu,'(e11.4e3)') x
x = 1.6732E7
write(iu,'(f0.1)') x
```

```
File contains:
12345
-.1532E+103
16732000.0
```

■ Character output

- variable length determines length of output for 'A' format without width specifier

■ On input

- use same format specifications as for writing
- note that F formatting in general behaves differently for input than for output (depends on input data) \rightarrow not dealt with in this course
- for strings, the length parameter determines how many characters are read if the 'A' format is used

Assumption:

- format string without components in parentheses
- more items in I/O list than edit descriptors are available

Output:

```
integer :: i(24)
i = ...
write(iu, '(10i4)') i
```

- will produce three records (the last one incomplete)
- format specification is repeated

Input:

- format exhaustion → remainder of record is **skipped**
- otherwise similar to output
- example: file with contents

```
 1  2  3  4  5
11 12 13 14 15
21 22 23
```

which is processed using

```
read(..., fmt='(3i3)') is(:3,:3)
```

will only read the values marked red (in which order?)

■ Exceptional case:

- format string **with** parenthesized components

■ Format processing:

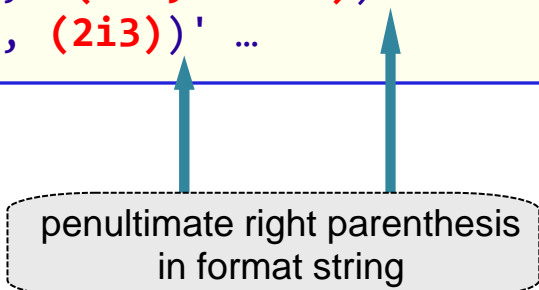
- when the last right parentheses are reached, select the format item enclosed by the parentheses whose right part precedes the last one
- include any repeat count associated with these parentheses

■ Examples:

- upon format exhaustion, control reverts to format items marked red

```
... fmt='(i4, 3(2i3,2e10.3))' ...  
... fmt='(i4, (2i3))' ...
```

penultimate right parenthesis
in format string



■ Perform I/O without conversion to character strings

- avoid conversion overhead
- avoid possible roundoff errors
- binary representation more space efficient

■ Requires suitable OPEN specification:

```
open(unit=iunit, &  
      action='WRITE', &  
      file='my_bin_file', &  
      form='UNFORMATTED')
```

■ Data transfer statements

- without format or namelist specification
- each transfer statement writes (or reads) exactly one record

```
write(iunit) x(1:n), y(1:n)
```

- processor may pad record to a convenient size
- reading a record must be performed consistently with the write (data type, array size, but order of array elements can be arbitrary)

■ Disadvantage: binary files may be unportable

- padding
- big- vs. little endian
- large file treatment

e.g. files or records larger than 2 GByte

■ Recommendations:

- may need to convert to formatted and back again
- if no derived type entities are written, intrinsic type representations are consistent and large files don't pose problems, then I/O on „foreign“ binary files may work anyway

■ Big- vs. little endian



- representation of intrinsic types differ only with respect to byte ordering
- compiler may offer switches and/or environment variables to deal with this situation

Now:
10 Minute break

■ General rules for all specifications

- a `unit=` or `newunit=` specifier is **required** for connections to external files
- a `file=` specifier supplying the name of the file to be opened must be provided under most circumstances
- character expressions on the RHS of a specification are often from a fixed list; these may be lower or upper case. Trailing blanks are ignored.

Table of additional OPEN specifications

mode keyword	argument values (defaults in bold)	semantics
access=	'direct', ' sequential ' or 'stream'	determines access method
action=	'read', 'write' or 'readwrite'	determines I/O direction; default is processor-dependent.
asynchronous=	'yes' or ' no '	 necessary (but not sufficient) for AIO
encoding=	' default ' or 'utf-8'	 UNICODE might work ...
form=	'formatted' or 'unformatted'	conversion method; default depends on access method.
position=	' asis ', 'rewind' or 'append'	specifies the initial position of the file (sequential or stream access)
recl=	positive integer value	record length (in file storage units – often 1 byte) for direct or sequential access files
status=	'old', 'new', ' unknown ', 'replace' or 'scratch'	enforce condition on existence state of file before the OPEN statement is executed.

■ General properties

- set additional properties in the OPEN statement which apply for **all** subsequent I/O statements
- set additional properties within subsequent READ or WRITE statements which apply for **that particular** statement
- use INQUIRE on unit to obtain presently set properties (see later; RHS expressions are then replaced by character string variables)
- these modes apply for formatted I/O **only**

■ Example:

```
real :: r = 2.33444e+2
open(iu, ...)
write(iu, '(e11.4)') r
write(iu, sign='plus', '(e11.4)') r
write(iu, sign='suppress' , '(e11.4)') r
```

assumption: open
does **not** specify **sign=**

expected output
b0.2334E+03
+0.2334E+03
b0.2334E+03

(first line is
processor dependent)

Table of changeable connection modes

mode keyword	argument values (defaults in bold)	semantics
blank=	' null ' or 'zero'	determine how blanks in input field are interpreted
decimal=	'comma' or ' point '	set character used as decimal point during numeric conversion
delim=	'apostrophe', 'quote' or ' none '	sets delimiter for character values in list-directed and namelist output
pad=	' yes ' or 'no'	padding with blanks during input if more characters must be processed than contained in record
round=	'up', 'down', 'zero', 'nearest', ' compatible ', 'processor_defined'	set rounding mode for formatted I/O processing
sign=	'plus', 'suppress' ' processor_defined '	controls whether an optional plus sign will appear in formatted output

■ Execution of CLOSE:

- terminates connection of previously OPENed file to specified unit
- at program termination, all connected units are implicitly CLOSEd
- application of CLOSE to a unit which does not exist or is not connected has no effect

■ status= specifier

- 'keep'
- 'delete'

default if OPENed with status other than 'scratch'

default if OPENed with status='scratch'

■ Notes:

1. 'keep' is not allowed if file was opened with status='scratch'
2. if 'keep' is specified for a non-existent file, it does not exist after execution of CLOSE

■ Obtain information about

- a **unit**'s connection properties („inquire by unit“), or
- connection properties allowed for a **file** („inquire by file“), or
- (minimum) **record length** needed for an output item („inquire by output list“ → see direct access file discussion)

■ General rules

- may specify a file or a unit, but not both
- uses inquiry specifiers of the form `keyword=variable`
- for some of the keywords (also those that are also permitted in an OPEN statement), an additional status of 'UNKNOWN' or 'UNDEFINED' may be returned

■ Inquiry on unit

```
character(len=12) :: fm, ac, bl
:
open(unit=22, action='READ', &
      file='my_file', &
      form='UNFORMATTED')

inquire(unit=22, form=fm, &
        action=ac, blank=bl)
```

if OPEN was successful:

```
trim(fm) has the value 'UNFORMATTED'
trim(ac) has the value 'READ'
trim(bl) has the value 'UNDEFINED'
```

- character values are returned in uppercase

■ Inquiry on file

```
character(len=12) :: fm
:
inquire(file='my_file', &
        form=fm)
```

- if my_file was not previously opened, trim(fm) has the value 'UNDEFINED'
- if it was opened before the INQUIRE using the statement from the left hand side of the slide, trim(fm) has the value 'UNFORMATTED'

Table of INQUIRE specifications specific to that statement

mode keyword	argument variable type (and possible return values)	semantics
direct=, sequential=, stream= F03	character string: 'YES', 'NO', or 'UNKNOWN'	determine whether specified access is allowed for file
exist=	logical	determine whether a file or unit exists
formatted=, unformatted=	character string: 'YES', 'NO', or 'UNKNOWN'	determine whether (un)formatted I/O is allowed
name=	character string	find the name of a file connected to a unit
named=	logical	find out if file has a name
nextrec=	integer	find the next record number of a direct access file
number=	integer	identify unit connected to a file (-1 if no unit is connected)
opened=	logical	determine whether file or unit is connected
read=, write=, readwrite=	character string: 'YES', 'NO', or 'UNKNOWN'	determine whether named access mode is allowed for file
size=	integer	determine size of a file (in file storage units; -1 if the size cannot be determined)

■ READ and WRITE statements

- allow the changeable connection mode specifiers already discussed for OPEN
- ... and we of course have seen the `unit` and `fmt` specifiers
- **additional** specifiers refer to specific I/O functionality which is discussed on the following slides (mostly by way of specific examples)

■ Note:

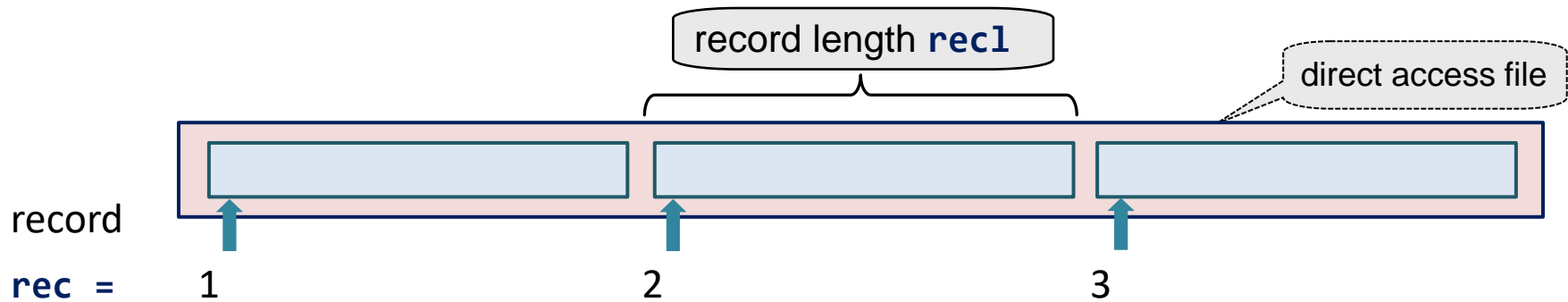
- Stream I/O
- Non-advancing I/O

are not dealt with in this course

Direct access files (1)

■ OPEN for direct access – differences to sequential files

- predefine file as a container with records of **equal** size
- records are identified by index number



- record size specified in file storage units (whose size is processor dependent)
- any record can be written, read and rewritten without interfering with another one
(contrast to sequential file: overwriting a record **invalidates** all following records)

■ A direct access file may be formatted or unformatted

- default is unformatted

Direct access files (2)

■ Step 1: determine maximum record size

- INQUIRE by output list may help

```
integer(kind=1k) :: max_length
inquire(iolength=max_length) &
  size(x), size(y), x, y
```

- specify complete I/O list
- objects should have the maximum size occurring during the program run

■ Step 3: Write a record

- record not filled → remainder is undefined

```
do nr=...
  : ! set up x, y
  write(unit=iu, rec=nr) size(x), size(y), x, y
end do
```

usually, a single record

■ Step 2: Create direct access file

```
open(unit=iu, file='da_file', &
  access='direct', &
  recl=max_length, &
  action='write', &
  status='replace')
```

- specify the **maximum** expected record length

... Step 4: close file

■ Open an existing direct access file for reading

```
inquire(file='my_da', recl=r_length)
open(unit=iu, file='da_file', access='direct', &
      recl=r_length, action='read')
do nr=...
  read(iu, rec=nr) nx, ny
  allocate(x(nx), y(ny))
  read(iu, rec=nr) nx, ny, x, y
  : ! process x, y
  deallocate(x, y)
end do
```

inquire by file

- information about number of records and the size of data to be read: „metadata“ that must be separately maintained (the latter, in the example, are written at the beginning of a record)

■ Limitations

- processor-dependent upper limit for record sizes (may differ for formatted and unformatted files)
- large number/size of records may lead to performance issues (access times)
- parallel access (MPI or coarray programs) to disjoint records may or may not work as expected (depends on access pattern and file system semantics)

■ Remark on formatted direct access

- slash edit descriptor causes record number to increase by one, and further I/O processing starts at the beginning of the next record

■ Part of state of connected file

- initial point established when connection is formed (OPEN) – at beginning of first record
- terminal point is just after last existing record

■ File position typically changes when either

- data transfer statements or
- positioning statements

are executed

■ Error conditions:

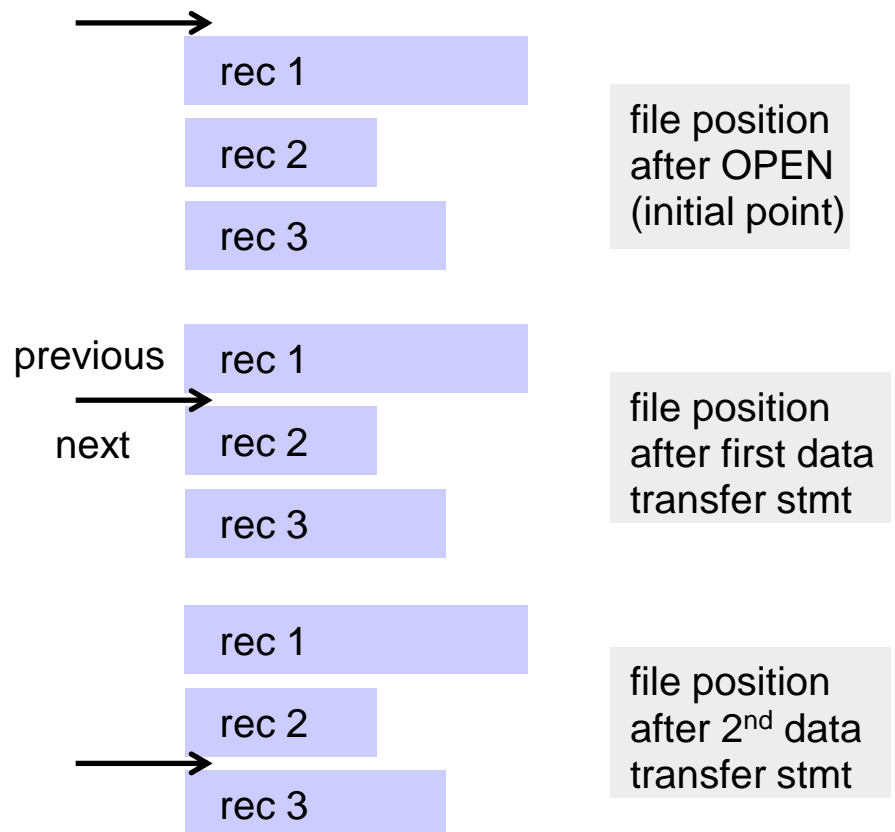
- lead to indeterminate file position

■ End-of-file condition:

- data transfer statement executed after terminal position was reached → abort unless END specifier present

■ Default I/O processing:

- „advancing“ → file position is always **between** records



■ BACKSPACE statement

- change file position to before the current record (if there is one), or else to before the previous record

```
backspace(<unit>)
```

beware
performance
issues

- the statement has no effect if the connection is in the initial position

■ ENDFILE statement

- write an EOF as the next record and position the file connection there

```
endfile(<unit>)
```

■ REWIND statement

- change position of file connection to initial position

```
rewind(<unit>)
```

- allows to revert from undefined to defined file position

■ Typically used

- for sequentially accessed files

■ An I/O statement may fail

■ Examples:

- opening a non-existing file with status='OLD'
- reading beyond the end of a file
- runtime error during format processing

■ Without additional measures, the RTL will terminate the program

■ Prevent this via user-defined error handling

- specify an **iostat** and possibly **iomsg** argument in the I/O statement
- legacy arguments: **err** / **end** (require a label to which execution branches) → do not use

■ Two logical functions

```
is_iostat_end(i)  
is_iostat_eor(i)
```

relevant for non-advancing
input only (not treated here)

are provided that check whether the **iostat** value of an I/O operation corresponds to an EOF (end of file) or EOR (end of record) condition

■ Graceful failure if the file `input.dat` does not exist

```
integer :: ios, iu
character(len=strmx) :: errstr
open(iu, file='input.dat', action='READ', form='FORMATTED', &
      status='OLD', iostat=ios, iomsg=errstr)
if (ios /= 0) then
  write(*,*) 'OPEN failed with error/message: ', ios, trim(errstr)
  error stop 1
end if
```

positive value returned in case of error

■ Gracefully dealing with an EOF condition

```
ioloop : do
  read(iu, fmt=..., iostat=ios, iomsg=errstr) x
  if (ios /= 0) then
    if (is_iostat_end(ios)) exit ioloop
    write(*,*) 'READ failed with error/message: ', ios, trim(errstr)
    error stop 1
  end if
  : ! process x
end do ioloop
```

negative value returned in case of EOF

■ 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

■ 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 namelist 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

■ What is an internal file?

- basically a character entity – a file storage area inside the program
- which replaces the unit number in data transfer statements

■ What is it used for?

- use the internal file as intermediate storage for conversion purposes e.g.,
 1. read data whose format is not known in advance („parsing“)
 2. prepare output lists containing a mixture of various data types

■ Example 1:

- represent an integer as string

```
character(len=range(1)+1) :: &
integer :: i
: ! define i
write(i_char, fmt='(i0)') i
write(*, fmt='(a)') &
    trim(i_char)
```

why this spec? → char

■ Rules:

- no explicit connection needed
- only formatted sequential access is possible
- explicit, list-directed and namelist formatting is possible

■ Rules (cont'd):

- file positioning and file inquiry are not available
- single record: corresponds to a character scalar
- multiple records: correspond to a character array
- length of string is the (maximum) record length

■ Example 2:

- generate format **dynamically**
- also illustrates character string descriptor

value of `my_fmt` is `'(7i4)'`

```
character(len=...) :: my_fmt
integer, allocatable :: iarr(:)
integer :: iw
: _____ iarr is allocated to size 7 and defined
iw = ... ! prospective width e.g., 4
write(my_fmt, fmt= &
      '('(' ',i0,'i',i0,')')' &
      ) size(iarr), iw
:
write(unit=..., fmt=my_fmt) iarr
```

Now proceeding to **last exercise session**



This concludes the workshop

Thanks for your attention!