An introduction to the Fortran programming language

Reinhold Bader
and
Gilbert Brietzke
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

<table>
<thead>
<tr>
<th>Fortran version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 66</td>
<td>Ancient</td>
</tr>
<tr>
<td>Fortran 77 (1980)</td>
<td>Traditional</td>
</tr>
<tr>
<td>Fortran 90 (1991)</td>
<td>Large revision</td>
</tr>
<tr>
<td>Fortran 95 (1997)</td>
<td>Small revision</td>
</tr>
<tr>
<td>Fortran 2003 (2004)</td>
<td>Large revision</td>
</tr>
<tr>
<td>Fortran 2008 (2010)</td>
<td>Mid-size revision</td>
</tr>
<tr>
<td>TS 29113 (2012)</td>
<td>Extends C interop</td>
</tr>
<tr>
<td>TS 18508 (2015)</td>
<td>Extends parallelism</td>
</tr>
<tr>
<td>Fortran 2018 (2018)</td>
<td>Next revision</td>
</tr>
</tbody>
</table>

TS $\rightarrow$ Technical Specifications

- „mini-standards“ targeted for future inclusion (modulo bug-fixes)
Conventions and Flags used in these talks

- **Standards conformance**
  - Recommended practice
  - Standard conforming, but considered questionable style
  - Dangerous practice, likely to introduce bugs and/or non-conforming behaviour
  - Gotcha! Non-conforming and/or definitely buggy

- **Legacy code**
  - Recommend replacement by a more modern feature (details are not covered in this course)

- **Implementation dependencies**
  - Processor dependent behaviour (may be unportable)

- **Performance**
  - Language feature for performance
Why Fortran?

- **SW engineering aspects**
  - good ratio of learning effort to productivity
  - good optimizability
  - compiler correctness checks
    (constraints and restrictions)

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

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

Some of the above are outside the scope of this course.
When not to use Fortran

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

- **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 F03 and F08 features
  - legacy features will be largely omitted (their existence might be noted)
  - content mostly targeted at new code development
Some references

- **Modern Fortran explained** (7th edition)
  - Michael Metcalf, John Reid, Malcolm Cohen. OUP, 2011

- **The Fortran 2003 Handbook**

- **Guide to Fortran 2008 Programming**

- **Download of (updated) PDFs of the slides and exercise archive**
  - freely available under a creative commons license
  - [http://www.lrz.de/services/software/programmierung/fortran90/courses/](http://www.lrz.de/services/software/programmierung/fortran90/courses/)
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
  \[ x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

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

    : executable statements: see next slide

    end program
```

© LRZ 2009-18
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

```fortran
program solve_my_quadratic
  implicit none
  real, parameter :: a = 2.0, b = -2.0, c = -1.5
  real :: x1, x2
  intrinsic :: sqrt
  x1 = ( -b + sqrt(b**2 - 4. * a * c) ) / ( 2. * a )
  x2 = ( -b - sqrt(b**2 - 4. * a * c) ) / ( 2. * a )
  write(*, fmt=*) 'Solutions are: ', x1, x2
end program
```

- **Execution order:**
  - Declarations must always precede executable statements
  - Executed in order of appearance

- **Keywords and concepts:**
  - Assignment
  - Intrinsic function call
  - Expression
  - I/O statement (output)
  - String literal
Compiling and running (simplest case)

Dependency:
- on processor (aka compiler) and operating system

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`

Execution of resulting binary
- `./prog.exe`

Compiled vs. interpreted code
- efficiency of execution
- typical speed factors: 20-60
- greatly care for large programs

will produce an output like
Solutions are: 1.50000000 -0.500000000

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

huge numbers of additional compiler options are typically available
## Invocations for various compilers

<table>
<thead>
<tr>
<th>Vendor (Platform)</th>
<th>most recent version</th>
<th>Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM (Power)</td>
<td>15.1</td>
<td>xlf, xlf2008, xlf2008_r</td>
</tr>
<tr>
<td>Intel (x86, x86_64)</td>
<td>18.0</td>
<td>ifort</td>
</tr>
<tr>
<td>PGI (x86, accelerators)</td>
<td>17.10</td>
<td>pgfortran, pgf90</td>
</tr>
<tr>
<td>GCC (many)</td>
<td>7.2</td>
<td>gfortran</td>
</tr>
<tr>
<td>NAG (many)</td>
<td>6.2</td>
<td>nagfor</td>
</tr>
<tr>
<td>Cray (Cray)</td>
<td>8.4</td>
<td>ftn</td>
</tr>
</tbody>
</table>
More on I/O

- **List-directed formatted output**

```plaintext
write(*,fmt=*) 'Hello'
```

same meaning as

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

or

```plaintext
print *, 'Hello'
```

- **Programmer control over layout:**

  - specify an explicit format string:

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

will produce output

```
Solutions are 1.50000 -0.50000E+00
AAAAAAAAAAAAAAAAAFFFFFFFFFFFFFXEEEEEEEEEEEEEEE
```

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

```
AAAAAAA
14 chars
```

```
12 chars, 5 decimals
```

```
12 chars, 5 decimals
```

( scientifc notation)
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)

Fixed source form
- avoid this legacy feature

---

© LRZ 2009-18
Introduction to the Fortran programming language
Technical reason for fixed source form ...

- A relic from an earlier age of computing: the punched card
Case insensitivity

For mostly historical reasons,

\[
\text{a} = 0.0 \\
\text{write}(*,*) \ 'Hello:\', \ a
\]

means exactly the same as

\[
\text{A} = 0.0 \\
\text{WRITE}(*,*) \ 'Hello:\', \ A
\]

Mixing upper and lower case is also permitted

However,

\[
\begin{align*}
\text{write}(*,*) & \ 'Hello' \\
\text{write}(*,*) & \ 'HELLO'
\end{align*}
\]

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

- **k_reverse**
- **q123**
- **Xx**

<table>
<thead>
<tr>
<th>Legal</th>
<th>Non-conforming</th>
</tr>
</thead>
<tbody>
<tr>
<td>k_reverse</td>
<td>1_fish</td>
</tr>
<tr>
<td>q123</td>
<td>a fish!</td>
</tr>
<tr>
<td>Xx</td>
<td>$sign</td>
</tr>
</tbody>
</table>

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

```fortran
integer :: day, month, year
```
Fortran’s type system

- **Recommendation:** Enforce strong typing ➔ all object’s types must be declared
  - avoid legacy implicit typing

- **Three numeric intrinsic types**
  1. integer
  2. real
  3. complex

- **Two non-numeric intrinsic types**
  4. character
  5. logical

- **Non-intrinsic types**
  - derived types will be discussed later

- **Numeric storage unit:** typical value nowadays 4 bytes

- **Character** :: `c = 's'`
- **Logical** :: `flag = .true.`, or `.false.`

- **Integer** :: `years`, `mass`
- **Complex** :: `psi`
Type parameters (1)

- An object declared `integer` can only represent values that are a subset of \( Z = \{0, \pm 1, \pm 2, \ldots\} \)
- typically \( \{-2^{32}, \ldots, +2^{32} - 1\} \)
- may be insufficient in some cases

- KIND type parameter
  - used for non-default representations:

```fortran
integer, parameter :: lk = selected_int_kind(16)
integer(kind=lk) :: seconds = 31208792336_lk
```

- value is not portable
- two storage units
- minimal decimal exponent
- \( 2^{32} \rightarrow 4,294,967,296 \)
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

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

minimal decimal digits and exponent

equivalent to previous statement
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()`

- **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

### Integer Kind

<table>
<thead>
<tr>
<th>integer kind</th>
<th>max. exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>$10^9$</td>
</tr>
<tr>
<td>extended</td>
<td>$10^{19}$</td>
</tr>
</tbody>
</table>

### Real Kind

<table>
<thead>
<tr>
<th>real kind</th>
<th>dec. digits</th>
<th>exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>6</td>
<td>$10^{-37} - 10^{+38}$</td>
</tr>
<tr>
<td>extended</td>
<td>15</td>
<td>$10^{-307} - 10^{+308}$</td>
</tr>
</tbody>
</table>

IEEE defined

---
Details on complex entities

- **Declaration:**
  ```fortran
  complex :: c
  complex(kind=kind(1.0d0)) :: z
  ```

- 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)
  ```fortran
  c = ( 1.2, 4.5e1 )
  z = ( 4.0_dk, 3.0_dk )
  ```

  where `dk` has the value `kind(1.0d0)`
Details on character entities (1)

- **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
Details on character entities (2)

- **String variables**
  - require *length* parametrization

```fortran
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.

```fortran
integer, parameter :: &
ck = kind('A')
character(kind=ck, &
  len=12) :: fh :
fh = ck_'Full House'
```

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

© LRZ 2009-18

Introduction to the Fortran programming language
Conditional execution (1)

Argument of $\sqrt{}()$

- a non-negative real number is required ("discriminant")
- replace executable statements by

```fortran
scalar logical expression
declared as real :: disc

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

General concept:

- "block construct" with one entry and one exit point
- modifies statement execution order
Conditional execution (2)

Repeated else if blocks:

```fortran
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:

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

- action statement: essentially any single statement
- examples:

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

legacy form of IF:
- arithmetic if
- not discussed here

"not equal"
Block diagram for conditional execution

Shown here:

- two conditions and an else block

Recommendation:
Do not omit else block
Avoid logically incomplete structure
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:**

```fortran
select case (index)
  case (0)
    x = 0.0
  case (1:4)
    x = 1.0
  case (5:)
    x = 2.0
  case default
    x = -1.0
end select
```

- no overlap is allowed within or between selectors → at most one block is executed
- here an **integer**
- single value
- selector
- lower and upper limits
- lower limit only
- no case fits (one block only)
Concept of an Array -
Simple array declaration and usage

**DIMENSION attribute:**

```fortran
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

- layout of (scalar) **array elements** in memory:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(1)</td>
<td>a(2)</td>
<td>a(3)</td>
<td>a(4)</td>
<td>a(5)</td>
<td>a(6)</td>
</tr>
</tbody>
</table>

- trivial mapping between storage sequence index and array index

- **lower bound**: a(1)
- **upper bound**: a(6)

**References and definitions of array elements: subscripting**

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

- Properties of subscript:
  - constant
  - variable
  - expression

- the above addresses **single array elements**
  → mechanisms to process complete arrays are needed

**Note:** there’s **much** more to array support in Fortran than this – stay tuned
Repeated execution (1) – the DO block construct

**Example:**
- Summing up the elements of an array:
  
  ```fortran
  s = a(1)
s = s + a(2)
  :s = s + a(6)
  ```

- Can be written more compactly as:
  
  ```fortran
  s = 0.
do i = 1, dm
    s = s + a(i)
  end do
  ```

**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.
Index set for general DO construct:

- if e3 is not specified, set e3=1
- start with e1 and increment by e3 as long as e2 not exceeded
- Empty index set:
- loop body is **not** executed; control of execution is transferred to statement after end of loop

Block diagram:

```
  i = e1
  i ≤ e2 ?
  no
  yes
    block
    i = i + e3
    continue
    regular
    execution
```

**legacy DO:**
- with labeled ending statement
- not discussed here
Nested block constructs
and fine-grain execution control

Example: nested loops

outer: do i=1,n
  do k=1,n
    calculate x
    if (x > 0.0) cycle outer
  end do
end do outer

• illustrates purpose of naming block constructs
• cycle statement forces starting the next iteration of the specified loop
  (without such a spec, that of the innermost enclosing loop)

Example: loop nested inside a BLOCK

ifound = 0
finder : block
  integer :: i
  do i=1,n
    if (x == a(i)) then
      ifound = i
      exit finder
    end if
  end do
end block finder

• exit statement proceeds to the end of the named block construct
  (may be loop or other construct)

BLOCK-local declaration is permitted
Repeated execution (3)

**Endless DO construct**

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

- requires a conditioned `exit` statement to eventually complete

**DO WHILE construct**

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

- 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
STOP and ERROR STOP

Syntax alternatives:

- `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 using coarrays, there is a difference)
Usage example: Intelligible error handling in main program unit

```
program solve_my_quadratic
  implicit none
  : disc = b**2 - 4. * a * c
  if (disc >= 0.0) then
    : calculate x1, x2
  else
    goto 100
  end if
  write(*,*) 'Solutions are: ', x1, x2
  stop 0
  100 continue ! error handling begins here
  write(*, fmt=*) 'Error: No real solution exists'
  error stop 1
end program
```

Makes use of the infamous labeled GOTO statement
- GOTO: branches execution
- CONTINUE: „do nothing“, serves as label carrier
- recommendation: except for this situation, **avoid** use of branching

Now we proceed to the **first exercise session** …
Model numbers, Expressions and Assignment
**Data representations**

**Numeric models for integer and real data**

- **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 \( w_k \in \{0, \ldots, r-1\} \)

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

- **real kind** is defined by:
  - positive integers \( p \) (digits), \( b > 1 \) (base, normally \( b = 2 \))
  - integers \( e_{\text{min}} < e_{\text{max}} \)

- **real value** is defined by:
  - sign \( s \in \{\pm 1\} \)
  - integer exponent \( e_{\text{min}} \leq e \leq e_{\text{max}} \)
  - sequence of \( f_k \in \{0, \ldots, b-1\} \), \( f_1 \) nonzero

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

or \( x = 0 \)

base 2 \( \Rightarrow \) „Bit Pattern“
### Inquiry intrinsics for model parameters

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Min Exponent(x)</th>
<th>Max Exponent(x)</th>
<th>Precision(x)</th>
<th>Radix(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>digits(x)</td>
<td>for real or integer x, returns the number of digits (p, q respectively) as a default integer value.</td>
<td></td>
<td></td>
<td></td>
<td>for real x, returns the default integer e\textsubscript{min}, e\textsubscript{max} respectively</td>
</tr>
<tr>
<td>precision(x)</td>
<td>for real or complex x, returns the default integer indicating the decimal precision (=decimal digits) for numbers with the kind of x.</td>
<td></td>
<td></td>
<td></td>
<td>for real or integer x, returns the default integer that is the base (b, r respectively) for the model x belongs to.</td>
</tr>
<tr>
<td>range(x)</td>
<td>for integer, real or complex x, returns the default integer indicating the decimal exponent range of the model x belongs to.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© LRZ 2009-18

Introduction to the Fortran programming language
Inquiry intrinsics for model numbers

**Example representation:** \( e \in \{-2, -1, 0, 1, 2\}, p=3 \)

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

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

**Mapping \( fl \):** \( \mathbb{R} \ni x \rightarrow \text{fl}(x) \)
- to nearest model number
- maximum relative error
  \[ \text{fl}(x) = x \cdot (1 + d), |d| < u \]
Typically used representations: IEEE-754 conforming

- matched to hardware capabilities

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

Negative zero:

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

- 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

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

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

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

- IEEE-754 adds
  - more rounding functionality
  - fulfills the standard req. above

There exist relevant algorithms for which less strict models cause failure!
Assignment to entities of intrinsic type

Simple example

```
real :: x

x = y * 2.0 + 3.0
```

**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)
Intrinsic assignment for arrays

Variant 1:
- LHS an array, RHS a scalar

```fortran
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

```fortran
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

- `real(rk) :: r` (4 bytes)
- `real(dk) :: d` (8 bytes)
- `integer(ik) :: i`
- `integer(lk) :: l`

Examples:

1. Not exactly representable values
   ```fortran
   r = 1.1
   d = r
   write(*,*) abs(d - 1.1_dk)
   ```

2. Rounding toward zero for real-to-integer conversion
   ```fortran
   r = -1.6
   i = r; write(*,*) i
   ```

3. Overflow (likely silent 🕳️)
   ```fortran
   l = 12345678900_lk
   i = l; write(*,*) i
   ```
Best practices for conversions

i. Use suitable intrinsics

ii. Limit conversion to the case stronger $\rightarrow$ 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)

2. Suitable intrinsic for real-to-integer conversion
   - r = -1.6
   - i = nint(r); write(*,*) i

3. Avoid overflow
   - if (abs(l) $\leq$ huge(i)) then
     - i = l
   - else
     - ! handle error
   - end if

© LRZ 2009-18
Introduction to the Fortran programming language
Conversion intrinsics

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmplx(x [, y] [, kind])</code></td>
<td>conversion to complex, or between complex KINDs</td>
</tr>
<tr>
<td><code>int(x [, kind])</code></td>
<td>conversion to integers, or between integer KINDs</td>
</tr>
<tr>
<td><code>real(x [, kind])</code></td>
<td>conversion to reals, or between real KINDs</td>
</tr>
</tbody>
</table>

Lots of further intrinsics exist, for example

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ceiling(a [, kind])</code></td>
<td>produces nearest higher (or lower) integer from real</td>
</tr>
<tr>
<td><code>floor(a [, kind])</code></td>
<td>produces nearest integer from real</td>
</tr>
<tr>
<td><code>nint(a, [, kind])</code></td>
<td>produces nearest integer from real</td>
</tr>
<tr>
<td><code>anint(a, [, kind])</code></td>
<td>produces nearest whole real from real</td>
</tr>
</tbody>
</table>

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

Expressions (1)

Operands and operators:

- **dyadic (or binary) operators:**
  
  \[
  \langle \text{operand} \rangle \ \text{operator} \ \langle \text{operand} \rangle
  \]

- **monadic (or unary) operators:**
  
  \[
  \text{operator} \ \langle \text{operand} \rangle
  \]

Combining binary and unary operators: In

\[
\langle \text{operand} \rangle \ \text{operator}_1 \ \text{operator}_2 \ \langle \text{operand} \rangle
\]

*\text{operator}_2* must be a unary operator
Expressions (2)

- **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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>also unary</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>also unary</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>slow</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>even slower</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>expression’s value:</th>
<th>6/3</th>
<th>8/3</th>
<th>-8/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>-2</td>
<td></td>
</tr>
</tbody>
</table>

- exponentiation with complex base: \( a^{\ast\ast}b \) produces principal value

\[ e^{b \cdot (\log|a| + i \ \text{arg}(a))} \]

with
\[ -\pi < \text{arg}(a) \leq \pi \]
Some examples for expression evaluation
(a, b, c, d of same numeric type and kind)

- Force order of evaluation by bracketing:
  
  \[
  \frac{a}{b}/c
  \]

  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)

- 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

  \[
  a ** (b ** c)
  \]

the general precedence and bracketing rules also apply for non-numeric operators.
Expressions (4): Mixed mode (numeric)

- **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
Expressions (5): Coercion table

**for a \( op \) b**

- with \( op \) one of the intrinsic numeric operations

<table>
<thead>
<tr>
<th>Type of a</th>
<th>Type of b</th>
<th>Coercion performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>R</td>
<td>a to R</td>
</tr>
<tr>
<td>I</td>
<td>C</td>
<td>a to C</td>
</tr>
<tr>
<td>R</td>
<td>I</td>
<td>b to R, except **</td>
</tr>
<tr>
<td>R</td>
<td>C</td>
<td>a to C</td>
</tr>
<tr>
<td>C</td>
<td>I</td>
<td>b to C, except **</td>
</tr>
<tr>
<td>C</td>
<td>R</td>
<td>b to C</td>
</tr>
</tbody>
</table>

**Special rules for exponentiation:**

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

\[ x**4 \rightarrow x***x*** \]

**Legend:**

- I → integer
- R → real
- C → complex
Expressions (6): Logical operations

Operands:
- variables as well as evaluated result are of type logical

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a.neqv.b</th>
<th>a.eqv.b</th>
<th>a.or.b</th>
<th>a.and.b</th>
<th>.not.a</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>unary</td>
</tr>
</tbody>
</table>

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

Examples:

```fortran
logical :: a, b, c, d
!
! define a, b, c
!
d = ( a .or. b ) .and. c
write(*,*) d
!
d = a .or. .not. c
write(*,*) d
```
Expressions (7): Relational operators

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

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

**Example:**

<table>
<thead>
<tr>
<th>F77</th>
<th>F95</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LT.</td>
<td><code>&lt;</code></td>
<td>less than</td>
</tr>
<tr>
<td>.LE.</td>
<td><code>&lt;=</code></td>
<td>less than or equal</td>
</tr>
<tr>
<td>.EQ.</td>
<td><code>==</code></td>
<td>equal</td>
</tr>
<tr>
<td>.NE.</td>
<td><code>=/</code></td>
<td>not equal</td>
</tr>
<tr>
<td>.GT.</td>
<td><code>&gt; </code></td>
<td>greater than</td>
</tr>
<tr>
<td>.GE.</td>
<td><code>&gt;=</code></td>
<td>greater than or equal</td>
</tr>
</tbody>
</table>

precedence: lower than numeric operators, higher than logical operators

logical :: r1, r2
real :: a
textured :: i, j
: ! define a, i, j

r1 = a >= 2.0

RHS

r2 = a < i - j

mixed mode expression: coercion is done as if sum were performed
Expressions (8): Character ordering

Collating sequence – a partial ordering
- \( A < B < \ldots < Y < Z \)
- \( 0 < 1 < \ldots < 8 < 9 \)
- either blank < \( A, Z < 0 \) or
  blank < \( 0, 9 < A \)

if lower-case letters exist:
- \( a < b < \ldots < y < z \)
- either blank < \( a, z < 0 \) or
  blank < \( 0, 9 < a \)

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

Various definitions are possible (e.g., ASCII, EBCDIC) → do not rely on a particular ordering
Expressions (9): Array constructor

- Establish complete set of array elements
  - with a single statement

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

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

- legacy notation (equivalent)

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

a = (/ 1.,3.,5.,7.,9.,11. /)
```
Character – Integer conversions

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

\[ \text{achar}(i) \]
\[ \{0, 1, 2, ..., 127\} \]
\[ \text{iachar}(i) \]
\[ \{a', b', ...\} \]

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

\[ \text{char}(i, \text{KIND}) \]
\[ \{0, 1, 2, ..., n - 1\} \]
\[ \text{ichar}(i) \]
\[ \{a', b', ...\} \]

KIND can be omitted for default character set
Alternative character ordering

- **Intrinsics 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

**Note:**
- zero-sized strings are identical

- \( \text{lge(string}_a, \text{string}_b) \): true if \( a \) follows \( b \) in the collating sequence or is equal, false otherwise
- \( \text{lgt(string}_a, \text{string}_b) \): true if \( a \) follows \( b \) in the collating sequence, false otherwise
- \( \text{lle(string}_a, \text{string}_b) \): true if \( b \) follows \( a \) in the collating sequence or is equal, false otherwise
- \( \text{llt(string}_a, \text{string}_b) \): true if \( b \) follows \( a \) in the collating sequence, false otherwise
Expressions (10): Character expressions

- 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:

```fortran
character(len=5) :: arg1, arg2
character(len=7) :: res1
character(len=12) :: res2

arg1 = 'no '  
arg2 = 'house'

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'
```

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

Now we proceed to the second exercise session …
Subprogram units
Separating out common tasks

- 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:

```fortran
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 \( n, x1, x2 \) from \( a, b, c \)
Subprogram code organization

Three organization variants are possible

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

---

© LRZ 2009-18

Introduction to the Fortran programming language
Module procedure

```fortran
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
```

- 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

```fortran
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
```

- The actual arguments `nsol` and possibly `x, y` are overwritten on each invocation

© LRZ 2009-18

Introduction to the Fortran programming language
Invoking a module procedure (2)

- From some other module procedure in the same module

```
A()
```

A statement in B() invokes A()

```
mod_solvers

solve_quadratic()
solve_something_else()

access by host association (no use statement)
```

- From a module procedure in another module

```
mod_solvers

solve_quadratic()
```

```
mod_other

other_procedure()
```

© LRZ 2009-18 Introduction to the Fortran programming language
Compiling multiple sources

- 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
  ```

  -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 system for mass production: (GNU) Make
Explicit interfaces

... 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.

This is the reason for the compilation order mentioned previously.

Mismatches cause rejection at compile time

„stand-alone“ procedures have an implicit interface.
→ checking is not possible
→ some language features will not work at all
Argument association
- each dummy argument becomes associated with its corresponding actual argument
- two variants:
  1. **Positional correspondence**
     
     ```fortran
     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
     
     ```fortran
     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
Procedure execution (2)

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 this is (essentially) 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
Declaring INTENT for dummy arguments

- Inform processor about expected usage

```fortran
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
```

- Semantics
  - effect on both implementation and invocation

<table>
<thead>
<tr>
<th>specified intent</th>
<th>property of dummy argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>procedure must not modify the argument (or any part of it)</td>
</tr>
<tr>
<td>out</td>
<td>actual argument must be a variable; it becomes <strong>undefined</strong> on entry to the procedure</td>
</tr>
<tr>
<td>inout</td>
<td>actual argument must be a variable; it retains its definition status on entry to the procedure</td>
</tr>
</tbody>
</table>

implies the need for **consistent** intent specification (fulfilled for module procedures)
Examples for the effect of INTENT specification

- **Compile-time rejection of invalid code**
  - subroutine implementation:
    ```fortran
    real, intent(in) :: a
    a = ... ! rejected by compiler
    ```
  - subroutine usage:
    ```fortran
    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
Functions – a variant of procedure

Example:

\[ w\text{sqrt}(x, p) = \sqrt{1 - \frac{x^2}{p^2}} \text{ if } |x| < |p| \]

```
module mod_functions
    implicit none
    contains
    real function wsqrt(x, p)
        real, intent(in) :: x, p
        wsqrt = ...
    end function wsqrt
end module
```

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
Using a RESULT clause

Alternative way of specifying a function result

- permits **separate** declaration of result and its attributes

```fortran
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:**

```fortran
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
```

- use of intrinsic logical function `present` is obligatory

**Invocations:**

```fortran
  y = wsqrt(x1, pg)  # uses path 1
  z = wsqrt(x2)       # 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.
Recursive procedures

- A procedure that invokes itself
  - directly or indirectly
    (may be a function or subroutine)
  
- requires the RECURSIVE attribute

- Example:
  - Fibonacci numbers

```fortran
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:
  - in F18, the recursive attribute will not be obligatory any more
Example:

```fortran
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 the main program or any kind of procedure, except an internal procedure).
- **Internal subroutine**
- **Internal function**
- **Invocation within host**
- `a, b` accessed from the host → **Host association**
- `slin` is accessed by **Host association**
Internal procedures (2)

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

- **Performance aspect**
  - if an internal procedure contains only a few executable statements, it can often be **inlined** by the compiler;
  - this avoids the procedure call overhead

- **Legacy functionality: statement function**

```fortran
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

- 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:

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

BLAS routine SSCALE sx ← sa * sx

- N elements processed
- INCX: stride between subsequent elements

real :: x(7)
: 
call sscal(4, 2.0, x, 2)
! overwrites “orange” elements

call sscal(3, -2.0, x(2), 2)
! overwrites “green” elements
The dangers of cheating …

… about the size of the actual argument

```fortran
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 \(\rightarrow\) 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
Character string dummy arguments

Assumed length string

```fortran
subroutine pass_string(c)
    intrinsic :: len
    character(len=*) :: c
    write(*,*) len(c)
    write(*,*) c
end subroutine
```

Usage:

```fortran
intrinsic :: trim
character(len=20) :: str
str = 'This is a string'
call pass_string(trim(str))
call pass_string(str(9:16))
```

- string length is passed implicitly
- produces the output

Now we proceed to the third exercise session …
Side effects in procedure calls
A simple example

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.)

- **Example for 3.:**

```fortran
program alias_host
  real :: x(5)
  call bar(x,5)
contains
  subroutine bar(this,n)
    real :: this(*)
    integer :: n
    ...
  end subroutine bar
end program
```

- inside bar(), this is aliased against x
A more subtle example

Procedure definition

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

... and invocation

```fortran
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)
Discussion of possible outcomes

**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) \( \rightarrow \) avoids aliasing

**Model 2: call-by-reference**
- pass address of memory location

- 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

© LRZ 2009-18

Introduction to the Fortran programming language
Aliasing restriction on dummy arguments

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)
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()`
Side effects of function calls

Example function

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

Undefined actual argument

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

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

Effective aliasing:

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

- 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
Dealing with side effects in function calls

- **Strategy 1:**
  - **document** proper usage
  - for the previous example, an invocation like
    \[ q = \text{badfun}(i) + \text{badfun}(j)^2 \]
    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)
Functions declared PURE

Example:

```fortran
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
- ...

→ 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
Subroutines declared PURE, etc.

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

Notes on PURE procedures in general:
- 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.
Passing arguments by value

- **Use VALUE attribute**
  - for dummy argument

- **Example:**

```fortran
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
```

- 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

```fortran
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
```

- Modified program that invokes the procedure

```fortran
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
```

- similarly, incorrect argument count is now caught by the compiler
- however, incorrect array size is usually not
Manually created interface for C library calls

- Additional language feature needed:
  - interoperability with C; intrinsic module ISO_C_BINDING

- Example: C function with prototype

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

- Fortran interface:

```fortran
module libm_interfaces
  implicit none
  interface
    real(c_float) function lgammaf_r(x, is) BIND(C)
    use, intrinsic :: iso_c_binding
  provides kind numbers for interoperating types
    real(c_float), value :: x
    integer(c_int) :: is
  end function
  end interface
end module
```

- Enforce C name mangling
- C-style value argument
Further comments on interoperability

**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
```

**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**
  
  \[ \int_{a}^{b} f(t) dt \]

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

- **Example:**
  - implementation of quadrature routine

```fortran
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
Procedures as arguments (2)

- Invoking the quadrature routine
  - step 1 – provide implementation of integrand

```fortran
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

```fortran
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
```
Abstract interface

- 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 the fourth exercise session …
Derived Types and more on Modules
Concept of derived type

- **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
contains
  ...
end module
```

- **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)

- **Type components:**
  - can be of intrinsic or derived type, scalar or array
  - further options discussed later

Layered creation of more complex types from simple ones
Structures

- Objects of derived type
- Examples:

```fortran
use mod_body

type(body) :: ball, copy

type(body) :: asteroids(ndim)
```

- Structure constructor
  - permits to give a value to an object of derived type (complete definition)
  ```fortran
  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
  ```fortran
  copy = ball
  ```
  - copies over each type component individually

- 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
Structures as dummy arguments

Implementation of „methods“

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

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)
Accessing type components

Via selector %

subroutine kick(this, dp)
  type(body), intent(inout) :: this
  real, intent(in) :: dp(3)
  integer :: i
  
  do i = 1, 3
    this % vel(i) = this % vel(i) + dp(i) / this % mass
  end do
end subroutine

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

(spaces are optional)
Remarks on storage layout

- **Single derived type object**
  - Compiler might insert padding between type components
  ```fortran
  type :: d_type
    character :: c
    real :: f
  end type
  ```
  Could look like: [Diagram showing storage layout]

- **Array element sequence**
  - As for arrays of intrinsic type
  ```fortran
  type(d_type) :: obj(3)
  ```
  [Diagram showing storage layout]

- **Special cases**
  - Sequence types **enforce** storage layout in specified order
    ```fortran
    type :: s_type
      sequence
      real :: f
      integer :: il(2)
    end type
    ```
  - **BIND(C)** types **enforce** C struct storage layout:
    ```fortran
    type, BIND(C) :: c_type
      real(c_float) :: f
      integer(c_int) :: il(2)
    end type
    ```
    Is interoperable with
    ```c
    typedef struct {
      float s;
      int i[2];
    } Ctype;
    ```
What is a module?

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

```
mod_body
  body%
  kick()
```

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

```fortran
set(...) 
my_data(:) = ...
```

```fortran
op1(...) 
... = my_data(:)
```

**my_data**

*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
```

**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**
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
```

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

```
use mod_globaldata
my_data(5) = ...
```

```
my_data is private \rightarrow rejected by compiler
```

```
call set(...) set() is public \rightarrow OK
```
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
```

- need to explicitly declare entities `public` that should be accessible by use association
Information hiding (3): Opaque derived types

Hide components

- 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

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

```fortran
use mod_date
type(date) :: easter
easter = date(2016,03,27)
components private – rejected by compiler
easter = set_date(2016,03,27)
public function set_date – OK
```
Information hiding (4): Mixed accessibility

Some type components PRIVATE, others PUBLIC

```fortran
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:

```fortran
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
The PROTECTED attribute

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

```fortran
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 of variable value only permitted in host association context

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

- read access is permitted
- modification OK because in host
- non-conforming – likely rejected by compiler
Propagation of use-associated entities

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)
Effect of PRIVATE on use-associated entities

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)
Name space issues

- **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

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

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

© LRZ 2009-18
Introduction to the Fortran programming language
119
Renaming module entities

Corrected example from previous slide

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

Avoiding naming collisions that result from use association only

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

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

Assumption:
- `mod_date` contains a public entity `lk`

Combine ONLY with renaming

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

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

- Modules are separately compiled

- If a program unit use associates a module
  - the latter must be compiled first
  - directed acyclic 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

© LRZ 2009-18
Introduction to the Fortran programming language
Typical implementation strategy

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

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)
Generating libraries

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

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

Now we proceed to the fifth exercise session …
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

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

- lower bounds: 1, -1
- upper bounds: 2, 1
- **shape**: 2, 3
  - i-th element of the shape is also called i-th **extent**
- **size**: 2 * 3 = 6
- layout in memory:

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

- "column major" array element sequence
  - bb(2,0) is fourth element in the sequence
  - 1st dimension
  - 2nd dimension
  - if no lower bound is specified, it has the value 1

Dimensions must be **constants** for "static" arrays.
Array inquiry intrinsic functions

**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**

- `shape(source)`

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

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

- Note: `extent = ubound - lbound + 1`
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

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

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

### Array assignment

- conformability of LHS and RHS: if RHS is not a scalar, shape must be the same
- scalars are broadcast
- element-wise assignment by array element order

```fortran
bb = d(4:5,16:18)
d(:,11:) = d(:,5:14)
```

- overlap of LHS and RHS → array temporary may be created

```fortran
real :: a(6)
we don't care about lower bounds here
```
Array sections (2): Vector subscripts

A rank 1 integer expression for subobject extraction

- one-to-one:

  \[
  \ldots = v( \begin{bmatrix} 2, 3, 9, 5 \end{bmatrix} )
  \]

- many-to-one:

  \[
  \ldots = v( \begin{bmatrix} 2, 3, 9, 2 \end{bmatrix} )
  \]

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

  \[
  iv = \begin{bmatrix} 2, 3, 9, 5 \end{bmatrix}
  \]

  \[
  \ldots = v( iv )
  \]

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.
Array sections (3): Zero size

- **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:**

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

- avoids the need for explicit masking
- remember array conformity rules
Array sections (4): rank reduction

Subarray formation may change rank of object:

```fortran
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
```

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

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

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

```fortran
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)
```
Array sections (5): derived types

- Earlier declarations ...

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

- Subobject designators:

<table>
<thead>
<tr>
<th>Designator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>asteroids(2)%pos(2)</td>
<td>real scalar</td>
</tr>
<tr>
<td>asteroids(2)%pos</td>
<td>real rank-1 array</td>
</tr>
<tr>
<td>asteroids(:)%pos(3)</td>
<td>real rank-1 array</td>
</tr>
<tr>
<td>asteroids(2)</td>
<td>scalar of type body (with array subobjects)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Designator</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>asteroids(:)%pos</td>
<td>disallowed</td>
</tr>
</tbody>
</table>
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!

```fortran
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)
```

elemental intrinsic

example:
a(6,11) is assigned the value b(1,1) * a(4,2)
Array intrinsics that perform reductions

<table>
<thead>
<tr>
<th>Name and arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all(mask [, dim])</td>
<td>returns .true. if all elements of logical array mask are true, or if mask has zero size, and .false. else</td>
</tr>
<tr>
<td>any(mask [, dim])</td>
<td>returns .true. if any element of logical array mask is true, and .false. if no elements are true or if mask has zero size.</td>
</tr>
<tr>
<td>count(mask [, dim])</td>
<td>returns a default integer value that is the number of elements of logical array mask that are true.</td>
</tr>
<tr>
<td>maxval(array [, dim] [, mask])</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name and arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minval(array [, dim] [, mask])</td>
<td>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.</td>
</tr>
<tr>
<td>product(array [, dim] [, mask])</td>
<td>returns the product of all elements of an integer, real or complex array. For a zero-sized array, one is returned.</td>
</tr>
<tr>
<td>sum(array [, dim] [, mask])</td>
<td>returns the sum of all elements of an integer, real or complex array. For a zero-sized array, zero is returned.</td>
</tr>
<tr>
<td>parity(mask [, dim])</td>
<td>returns .true. if .neqv. of all elements of logical array mask is true, and .false. else.</td>
</tr>
</tbody>
</table>

**Eight transformational functions**
- except for **count**, result is of same type and kind as argument

**Additional optional arguments**
- provide extra semantics
- see following slides
Optional argument \texttt{dim}

- Perform reduction along a single array dimension
  - other dimensions are treated elementally ($\rightarrow$ result is an \texttt{array})
  
  \begin{verbatim}
  real :: x(6,4)
  real :: xs2(6)
  : ! define x
  xs2 = sum(x, dim = 2)
  \end{verbatim}

- example above: \(xs2(i)\) contains \(\text{sum}(x(i,:))\)
- \texttt{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:

\[
\sum_j x(:,j)
\]

- argument array \texttt{x}
- result array (assigned to \texttt{xs2})

© LRZ 2009-18

Introduction to the Fortran programming language
Optional argument \texttt{mask}

Select a subset of elements

- some functions may use a \texttt{logical} array \texttt{mask} as a third optional argument
- \texttt{mask} must have same shape as the first argument

```fortran
real :: a(4), s
a = ...
s = sum(a, \texttt{mask} = a>0.)
```

Combining \texttt{dim} and \texttt{mask}

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

Further intrinsics that support \texttt{dim} and \texttt{mask} exist

- see compiler documentation

Illustration of masked reduction

- argument array \texttt{a}
- result scalar \texttt{s}
Array location intrinsics

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

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

**Example:**
```fortran
integer :: x(2,-1:1)
x = reshape([2,3,5,1,1,1], &
             shape=[2,3])
write(*,*) maxloc(x)  12
write(*,*) maxloc(x, dim=2)  21
```

- lower bounds remapped
- array element values
## Transformational array intrinsics

<table>
<thead>
<tr>
<th>Name and arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dot_product</strong> (vector_a, vector_b)</td>
<td>dot (scalar) product of numerical or logical rank 1 arrays.</td>
</tr>
<tr>
<td><strong>matmul</strong> (matrix_a, matrix_b)</td>
<td>matrix multiplication of numeric arrays of rank 1 or 2</td>
</tr>
<tr>
<td><strong>transpose</strong> (matrix)</td>
<td>transposition of rank 2 array representing a matrix</td>
</tr>
<tr>
<td></td>
<td>matrix(i, j) (\rightarrow) matrix(j, i)</td>
</tr>
<tr>
<td><strong>merge</strong> (tsource, fsource, mask)</td>
<td>elemental merging of two arrays of same type and shape, based on logical mask value</td>
</tr>
<tr>
<td><strong>spread</strong> (source, dim, ncopies)</td>
<td>replicate an array <strong>ncopies</strong> time along dimension <strong>dim</strong></td>
</tr>
<tr>
<td><strong>reshape</strong> (source, shape [, pad] [, order] )</td>
<td><strong>reshape</strong> optional arguments:</td>
</tr>
<tr>
<td></td>
<td>• <strong>pad</strong> array to fill in excess elements of result</td>
</tr>
<tr>
<td></td>
<td>• subscript permutation via integer permutation array <strong>order</strong></td>
</tr>
<tr>
<td><strong>cshift</strong> (array, shift [, dim])</td>
<td>circular shift of array elements along dimension 1 or <strong>dim</strong></td>
</tr>
<tr>
<td><strong>eoshift</strong> (array, shift [, boundary] [, dim])</td>
<td>end-off shift of array elements along dimension 1 or <strong>dim</strong>, using <strong>boundary</strong> to fill in gaps if supplied</td>
</tr>
</tbody>
</table>
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
Some comments on current hardware

- **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 → transfer of memory block ("cache line", CL) from memory

- **Cache fills up ...**
  - usually least recently used CL is evicted

**Example:**
\[
c(\cdot) = a(\cdot) + \ldots
\]
Serial vs. parallel execution

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. availability

Parallel models
- inside Fortran: DO CONCURRENT Coarrays
- outside Fortran: Library approach (MPI) Directive approach (OpenMP)

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 $\Delta T$ between

- request of worker for single datum

and

- availability of data item for being worked on

depends on speed and length of assembly line
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**!

storage: RAM, remote node, disk, ...

Now give me **all**!

All items delivered
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

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

High bandwidth available

Low bandwidth available

Data start out residing here
Using synthetic loop kernels for performance evaluation

**Characteristics**

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

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Name</th>
<th>Flops</th>
<th>Loads</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s = s + a_i \cdot b_i )</td>
<td>Scalar Product</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( n^2 = n^2 + a_i \cdot a_i )</td>
<td>Norm</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( a_i = b_i \cdot s + c_i )</td>
<td>Linked Triad (Stream)</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( a_i = b_i \cdot c_i + d_i )</td>
<td><strong>Vector Triad</strong></td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

- run repeated iterations for varying vector lengths (working set sizes)
Vector Triad $D(:, :) = A(:, :) + B(:, :) \times C(:, :)$  

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

- **L1D** – 32kB  
  - < 112 GB/s

- **L2** – 256kB  
  - < 62 GB/s

- **L3** – 20MB  
  - ~ 33 GB/s

Memory  
  - ~ 14.7 GB/s

**measured „effective“ BW:**  
- 3 LD+1ST  
- 16 Bytes / Flop, repeated execution  
  (actually issued: 4 LD+1ST in L2 and higher)

Vectorization (256 Bit registers) provides performance boost mostly in L1/L2 cache
Performance by type and kind

Sandy Bridge 2.3 GHz with AVX / ifort 16.0

- double prec
- single prec
- 16 Byte real
- default int
- large int
- double cmplx

uses SSE 4.1 VEX

working set size is different for same vector length

~ 60 MFlop/s

Vector length
Hardware dependence of Triad Performance

Double Precision Triad

Note:
vector processors have a qualitatively different characteristic
Microprocessor Architecture continued

**Loads and Stores**
- apply to cache lines

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

**Pre-Requisite:**
- spatial locality
- violation of spatial locality:

  if only part of a cache line is used → effective reduction in bandwidth observed
Performance of strided triad on Sandy Bridge
(loss of spatial locality)

\[ D(:,:,\text{stride}) = A(:,:,\text{stride}) + B(:,:,\text{stride}) \times C(:,:,\text{stride}) \]

Notes:
- stride known at compile time
- serial compiler optimizations may compensate performance losses in real-life code

Example: stride 3

Graph showing performance (MFlop/s) vs. stride for different data sizes (N=1024, 8192, 8388608).

- 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
```

Correct:

```
real :: a(ndim, mdim)
do j=1, m
   do i=1, n
      a(i, j) = ...
   end do
end do
```

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

Correct:

```
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
Fortran language features targeting performance

- 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
After the Lunch break ...

Fortran Environment
## Intrinsics

### Processing the command line

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>command_argument_count()</code></td>
<td>Integer function that returns what it says</td>
</tr>
<tr>
<td><code>get_command_argument( number [, value] [, length] [, status] )</code></td>
<td>Subroutine that delivers information about a single command line argument</td>
</tr>
<tr>
<td><code>get_command( command [, length] [, status] )</code></td>
<td>Subroutine that delivers information about the complete command line</td>
</tr>
</tbody>
</table>

### Executing system commands

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execute_command_line( command [, wait] [, exitstat], [, cmdstat] [, cmdmsg] )</code></td>
<td>Subroutine that executes a system command specified as a string</td>
</tr>
</tbody>
</table>

Replaces the non-standard extension

```
call system(command)
```

### Process environment variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_environment_variable( name [, value] [, length] [, status] [, trim_name] )</code></td>
<td>Subroutine that delivers information about a named environment variable</td>
</tr>
</tbody>
</table>
Obtain the value of the PATH variable:

```fortran
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
## Intrinsic Module ISO_FORTRAN_ENV

- **Contains some often-used constants**
  - Here a subset:

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

- **Contains some inquiry procedures**
  - `compiler_options()`
  - `compiler_version()`
  - return string constants

- **Some of this was added in** F08

Usage examples

```fortran
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
Examples for nested scoping (1)

### 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
```

- **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.
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 \texttt{is} is a „class 3“ identifier (must be unique per-interface); it is a separate entity that overrides global \texttt{is}

local \texttt{js} in scope 2; it is a separate entity that overrides global \texttt{js}

\texttt{ks} is host associated from scope 1

\texttt{is} and \texttt{js} are host associated from scope 2

\textbf{source of programming errors or performance problems}

- forget local declaration for entities meant to be local \rightarrow access by host association
- \texttt{implicit none} does not help here
- less probable if suggestive names are given to globals
Lifetime of local and global entities

**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
Explicit SAVE attribute

**Example**
- (legacy) standalone procedure

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

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

  save :: work

  if (first .EQ. 0) then
    ...
    work(...) = ...
    first = 1
  end if
  update array a
end subroutine
```

**Properties:**
- at the subsequent invocation of the procedure, SAVEEd 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 SAVEEd 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
What are initializations?

- **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
Initialization of variables (1)

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

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

**Variant 2:**
- the DATA statement

```fortran
integer :: i
character(len=4) :: cn
type(date) :: o
type(date) :: o = date(…)
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
Initialization of variables (2)

**Consequences:**

- Initialized variables acquire the (implicit) SAVE attribute.
- **Different** from C semantics (similar syntax!)

**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:** 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

```fortran
... ( <expr>, i = low, high[, step] ) ...
```

- loop index, low, high and step values are integers of any kind

- 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)
Implied-do loops (2)

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
Implied-do loops (3)

Examples for scenario 2

```fortran
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
```

- 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

```fortran
  type(person) :: chairman
  write(*,*) chairman % name
  write(*,*) chairman % birthday
  chairman = person(location = 'Room 23')
```

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
Specification inquiries and functions

- **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

---

**Introduction to the Fortran programming language**

© LRZ 2009-18
Examples

Function returning a string

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

Not permitted:
- non-constant expression in main program or module spec. part

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

→ compiler throws error

Declare working space
- **automatic** (non-SAVEEd!) variables

```fortran
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) )
        : 
    end subroutine proc
    pure integer function sfun(n)
        integer, intent(in) :: n
        sfun = dm * n + da
    end function sfun
end module mod_proc
```
Notes on automatic objects

- 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 the sixth exercise session …
Array Processing – Part 2

Procedure interfaces and block constructs
Assumed shape dummy argument

This is the recommended array argument style

```fortran
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
```

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

  ```fortran
  real :: ad(:,:)
  ```
Invocation is straightforward

```fortran
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
```

Actual argument
- must have a shape
- can be an array section

Normally, a descriptor will be created and passed → no copying of data happens

access explicit interface for `process_array`

consistency of argument’s `type, kind` and `rank` with interface specification is required
Memory layouts for assumed shape dummy objects

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

  - Remapped lower bound
  
  - `dim_2`:
    - Indicates array element sequence of dummy argument
  
  - `is_contiguous(ad)` returns `.true.`

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

  - `dim_2`:
    - All "orange" storage units are not part of the dummy object. They are **invisible**.
  
  - `is_contiguous(ad)` returns `.false.`
The CONTIGUOUS attribute

For large problem sizes,
- non-contiguous access inefficient due to loss of spatial locality

```fortran
module mod_solver
  implicit none
contains
  subroutine process_array_contig(ad)
    real, intent(inout), contiguous :: ad (:,:)
    : end subroutine
end module
```

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
...
```

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: \( LDA \times (M-1) + N \) 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

```fortran
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

AD == AA

part of array actually defined in procedure call
Assumed size: non-contiguous actual argument

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

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

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

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

\[ N = \text{LDA} \]

Array \( AD \) is completely defined in procedure call.

\[ \text{i.e., size( } aa(1:2*n:2,:), 1 \text{ )} \]
Assumed size: rank mismatch

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

```fortran
integer, parameter :: aadim = ...
real :: aa(aadim)

! calculate lda, n, m
call slvr( aa, lda, n, m )
```

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

- blocked processing of subarrays

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

- 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
Explicit shape dummy argument

- **Dummy array bounds**
  - declared via specification expressions

  ```fortran
  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)
Array-valued functions

Example:

```fortran
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

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

Declaration:

- **elemental** prefix:

```fortran
d module elem_stuff
c contains

elemental subroutine swap(x, y)
  real, intent(inout) :: x, y
  real :: wk
  wk = x; x = y; y = wk
end subroutine swap
end module
```

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

F08 introduces an IMPURE attribute for cases where PURE is inappropriate
Invoking an ELEMENTAL procedure

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

```fortran
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.,

    ```fortran
    where ( a > 0.0 ) a = 1.0/a
    ```

  - general form:

    ```fortran
    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

    ```fortran
    where ( x )
    y1 = ...
y2 = ...
y3 = ...
    [ elsewhere [( z )]
y4 = ... ]
    end where
    ```

    same mask applies for every assignment

  - `y4` is assigned for all elements with `.not. x .and. z`

© LRZ 2009-18
Introduction to the Fortran programming language 194
Assignment and expression in a WHERE statement or construct

Assignment may be
- a defined assignment (introduced later) if it is elemental

Right hand side
- may contain an elemental function reference. Then, masking extends to that reference
- may contain a non-elemental function reference. Masking does not extend to the argument of that reference

```
where (a > 0.0) &
  a = sqrt(a)
```

**sqrt()** is an elemental intrinsic

```
where (a > 0.0) &
  a = a / sum(log(a))
```

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

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

Parallel semantics

- of array element assignment

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

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

- conditional array element assignment

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

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

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

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

```fortran
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
```

- **Semantics**: each statement is executed for all index values **before**
  the next statement is initiated
  in the example, the third statement is conforming if a(:,m) was defined prior to the
  FORALL construct; the other values of a are determined by the first statement.

- this limits parallelism to each individual statement inside the block
Further notes on FORALL

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

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

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

will flag FORALL obsolescent
The DO CONCURRENT construct

**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
```

- 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
Examples

Incorrect usage

```
do concurrent (i=1:n, j=1:m)
  x = a(i, j) + ...
  b(i, j) = x * c(j, i)
  if (j > 1) a(i, j) = b(i, j-1)
end do
```

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

Correct usage

```
do concurrent (i=1:n, j=1:m)

  block
    real :: x
    x = a(i, j) + ...
    b(i, j) = x * c(j, i)
  end block
end do
```

```
do concurrent (j=2:m)
  a(:, j) = b(:, j-1)
end do
```

Now we proceed to the seventh exercise session …
Dynamic Entities and Memory management
Some remarks about memory organization

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

**Virtual memory**

- high address
- executable code (non-writable)
- initialized global variables
- uninitialized global variables („block started by symbol“)

**static memory**

<table>
<thead>
<tr>
<th>Stack: dynamic data needed due to generation of new scope (grows/shrinks automatically as subprograms are invoked or completed; size limitations apply)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Heap: dynamically allocated memory (grows/shrinks under explicit programmer control, may cause fragmentation)</th>
</tr>
</thead>
</table>

Introduction to the Fortran programming language
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**

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

**Allocation, use and deallocation**

```fortran
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 )
```

*phi* is used for calculations ...

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

- 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 objects (2)

Allocatable scalars

- this feature allows to determine size of character strings at run time (making the use of ISO_VARYING_STRINGS mostly obsolete)
- otherwise relevant for polymorphic objects (not dealt with in this course)
Rules for allocatable objects

- **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:

```fortran
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)
Allocatable objects (3): Auto-allocation

Assignment to allocatable object

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

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

```fortran
ai = [ 1, 2 ]
str = "Hello World"
```

- Re-allocation to new size / string length

```fortran
ai = [ 3, 4, 7 ]
str = str(1:5)
```

- No re-allocation because RHS is conformant

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

No reallocation happens with an array section LHS: shape conformance is programmer’s responsibility

compiler switches are usually available to revert to 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

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

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 ...

---

© LRZ 2009-18

Introduction to the Fortran programming language
The POINTER and TARGET attributes

Declaration:

```fortran
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:

```fortran
pt => tg
```

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

**Example:**

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

pt => tg

pt(2) = 7.2

pt => null()
```

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

**Symbolic representation**

- **pt**
  - state after declaration: **undefined**
  - after pointer assignment: **associated**
  - **disassociated** after nullification

- **tg**
  - **association to target** via alias
  - **7.2**
Creation of an **anonymous** TARGET

```fortran
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 )
```

- The 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.

**Deferred shape → shape determined at run time**

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
The ASSOCIATED intrinsic

A logical function that
- returns association status of an entity with POINTER attribute;
- it cannot be applied to an undefined POINTER

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

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

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

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

prints T (.TRUE.), twice
prints T (.TRUE.)
prints F (.FALSE.)
prints F (.FALSE.)
Aliasing of subobjects

Subobjects of a target
- also are targets

Example:

```fortran
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

- After first assignment:
  - Only orange parts are aliased

- After second assignment:
  - Only orange parts are aliased

`is_contiguous(pt)` returns `.false.`
Avoid the initially undefined state

- `null()` intrinsic function \(\rightarrow\) start with disassociated state

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

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

```fortran
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)
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
deallocation

pt is nullified
pt_2 has same target as pt

... but pt_2 is undefined
(cannot use associated on it)
Container types

Allocatable type components

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

- a "value" container
- default (initial) value is not allocated

POINTER type components

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

- a "reference" container
- default value is disassociated
- example type is self-referential \( \rightarrow \) "linked list"

Note: Container types will not be thoroughly treated in this course
### Container types (2): Object declaration and assignment semantics

#### Allocatable type components

```fortran
module example_mod
  implicit none
  type (polynomial) :: p1, p2

  p2 = p1
end module example_mod
```

- **Assignment statement** is equivalent to
  ```fortran
  if ( allocated(p2%f) ) &
      deallocate(p2%f)
  allocate(p2%f(size(p1%f)))
  p2%f(:) = p1%f
  ```

- „**deep copy**“

#### POINTER type components

```fortran
module example_mod
  implicit none
  type (sparse) :: s1, s2

  s2 = s1
end module example_mod
```

- **Assignment statement** is equivalent to
  ```fortran
  s2%index = s1%index
  s2%value = s1%value
  s2%next => s1%next
  ```

- „**shallow copy**“

A reference, not a copy
Allocatable type components

- \( \text{type(T) :: } \text{p1} \)
  
  \( \text{p1 = T( [1.0, 2.0] )} \)

- dynamically allocates \( \text{p1}\%f \) to become a size 2 array with elements 1.0 and 2.0

When object becomes undefined

- allocatable components are automatically deallocated

POINTER type components

- \( \text{type(T) :: } \text{s1} \)
  
  \( \text{s1 = T( 3, 1.0, \text{null()} )} \)

- alternative:
  
  \( \text{s1 = T( 3, 1.0, \text{t1} )} \)

- not permitted:
  
  \( \text{s1 = T( 3, 1.0, \text{t2} )} \)

- a constant cannot be a target

\( \rightarrow \) e.g., overload constructor to avoid this situation (create argument copy)
Container types (4): Storage layout

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

```fortran
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)

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

POINTER dummy argument
- example: handling of a „reference container“

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

Actual argument must have matching attribute

© LRZ 2009-18

### INTENT semantics for dynamic objects

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

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

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

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

    What arrives inside `add_reference`?
    ```fortran
    subroutine add_reference(...) :
        if (associated(item)) a_container%item => item
    ```

    - this is different from assumed-shape, where bounds are remapped
    - it applies for both POINTER and ALLOCATABLE objects
    
    **Explicit (lower) bounds remapping is possible:**
    ```fortran
    if (associated(item)) a_container % item(1:) => 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
CONTIGUOUS pointers

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

```fortran
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:

```fortran
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

Possible invocations:
- efficient (uses auto-allocation on assignment):
  ```fortran
  integer, allocatable :: res(:)
  res = deduplicate(array)
  ```
- less efficient (two function calls needed):
  ```fortran
  integer :: res(ndim)
  res(:size(deduplicate(array))) = deduplicate(array)
  ```
- function result is auto-deallocated after completion of invocation
POINTER function results
(explicit interface required)

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

**Example: filtering a list**

```fortran
function next_uppertr(s, i) result(r)
  type(sparse), intent(in) :: s
  integer, intent(in) :: i
  type(sparse), pointer :: r
  r => null()
  do
    if (associated(s%next)) then
      r => s%next
      if (r%index <= i) exit
    end if
  end do
  if (r%index > i) r => null()
end function next_uppertr
```

**invocation:**

```fortran
type(sparse), target :: trm(nd)
type(sparse), pointer :: entry :
  set up trm (linked list)
do i=1, nd
  entry => trm(i)
do while ( associated(entry) )
  entry => next_uppertr(entry,i)
end do
```

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

© LRZ 2009-18

Introduction to the Fortran programming language
Opinionated recommendations:

- **Dynamic entities should be used, but sparingly and systematically**
  - performance impact, avoid fragmentation of memory → allocate all needed storage at the beginning, and deallocate at the end of your program; keep allocations and deallocations properly ordered.

- **If possible, ALLOCATABLE entities should be used rather than POINTER entities**
  - avoid memory management issues (dangling pointers and leaks)
  - especially avoid using functions with pointer result
  - aliasing via pointers has additional negative performance impact

- **A few scenarios where pointers may not be avoidable:**
  - information structures → program these in an encapsulated manner (see later for how to do that): user of the facilities should not see a pointer at all, and should not need to declare entities targets.
  - subobject referencing (arrays and derived types) → performance impact (loss of spatial locality, suppression of vectorization)!
Some more vector triad performance data...

```fortran
do i = 1, nsize
  d(i) = a(i) + b(i) * c(i)
end do
```

Sandy Bridge 2.3 GHz with AVX / ifort 15.0

- **double prec**
- **array syntax**
- **pointer, intent(in)**
- **pointer, intent(in), contiguous**
- **pointer, intent(inout)**
- **assumed shape**
- **assumed shape, contiguous**

procedure dummy arguments, processed with a DO loop
Object-based processing of the triad

**Candidate 1 (LoA):**
- "list of arrays"
- vary static block size \texttt{idim}

```
type LoA
    real(rk) :: x(idim)
type(LoA), pointer :: &
    next => null()
end type
```

**Candidate 2 (SoA):**
- "structure of arrays"
- length-parameterized derived type

```
type SoA (idim)
    integer, len :: idim
    real(rk) :: x(idim)
end type
```

**Candidate 3 (AoS):**
- "array of structures"
- knowledge of processing sequence goes into type design
- not always feasible (and neither advisable – remember Don Knuth's dictum)
- heavily changing access patterns undermine this concept
- alignment issues for components of varying type
LoA performance for varying block size

LoA on Sandy Bridge 2.3 GHz with AVX / ifort 16.0

Spatial locality is partially lost.
Optimal block size for large datasets: between 256 and 512
Comparing performance for remaining variants (ifort 17.0.1 on SandyBridge)

immature implementation? compiler should vectorize ...

Following now: 8th exercise session
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}} \text{ 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

```fortran
module mod_functions
  interface wsqrt
    procedure wsqrt
    procedure wsqrt_dk
  end interface
  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
```

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

also permitted:
- `module procedure` only expose the generic name
- `dk` specifies non-default real
Generic Interfaces (2)

**Invocation**

```fortran
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 \(\rightarrow\) rejected by compiler
```

**Specific functions**

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

**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

© LRZ 2009-18

Introduction to the Fortran programming language
Generic Interfaces (3): Keyword call

- **The following generic**
  (which legitimately references interfaces of external procedures)

```fortran
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

```fortran
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)
Exception to naming rules

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

Unambiguous resolution:

- also depends on which specifics are accessed
- \textit{gfun}(): interface of \textit{fun\_spec2()} might be ambiguous with respect to other specifics (not recommended!), since not use associated by „prog“
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

New feature in Technical Specification 29113: assumed rank
- only one specific is needed
- designed for C-interop → object can currently not be accessed in Fortran
- used e.g. for MPI implementations
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
Notes on overloading the structure constructor

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

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

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

o_d1 = date(12, 10, 2012)

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

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)
Operator overloading

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
Extending intrinsic operators

Example: add fractions

```fortran
module rational
  : restricted named interface
  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)
    type(fraction) :: r
    end function
end module
```

Usage:

```fortran
use rational

type(fraction) :: x, y, z

x = y + z

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
**Example: convolution**

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

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

**Usage:**

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

: define x, op
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
Expressions involving overloaded operators

- **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):
    \[ X = (A \times B) \times C; \ Y = A \times (B \times C) \]
  - 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 `*`
  \[ X = A + B \times C \]
is by default evaluated as
  \[ X = A + (B \times C) \]
Expressions involving defined operators

- **Unary defined operators**
  - have *higher* precedence than any other operator

- **Binary defined operators**
  - have *lower* precedence than any other operator

**Parentheses may be vital**

- The expression
  
  \[ X = A .\text{convolve}. \ (B + C) \]

  is evaluated as
  
  \[ X = A .\text{convolve}. \ B + C \]

  which very probably is not what you meant.

- What you meant must be written
  
  \[ X = (A .\text{convolve}. \ B) + C \]
Further properties of generic interfaces

- **Renaming of defined operators on the USE line**
  
  ```fortran
  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
Some limitations of default assignment

Example: arrays of type(sparse)

```fortran
type(sparse), allocatable :: A(:), B(:)
: ! set up A
B = A
```

- B(i)%next has same target as A(i)%next
- issue is caused by POINTER component

Default assignment is unavailable between objects of different derived types
Overloading the assignment operator

- Uses a restricted named interface:

```fortran
module mod_sparse
  : ! type definition of sparse
  interface assignment(=)
    procedure assign_sparse
  end interface
contains
  subroutine assign_sparse(res, src)
    type(sparse), intent(out) :: res
    type(sparse), intent(in) :: src
    : implement a deep copy
  end subroutine
end module
```

- create a clone of the RHS

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

**Quiz:** what might be missing in the procedure definition?
Overloaded assignment of function results:

Dealing with POINTER-related memory leaks

**Scenario:**
- RHS may be a function value (e.g. an expression involving a defined operator, or an overloaded constructor)

```
A(i) = sp_func(...)  
```

```
next  
```

Deep copy of component

Becomes orphaned → potential leak

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

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

**Now: 9th exercise session**
(may want to look at optional slides that follow)
Finalizing procedure implementation

**elemental recursive subroutine finalize_sparse**(this)

```fortran
  type(sparse), intent(inout) :: this
  if (associated(this%next)) then
    deallocate(this%next)
  end if
end subroutine
```

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

**Quiz:** what happens in the assignment

```
A(i) = sp_func(...)
```

if a finalizer is defined, but the assignment is not overloaded?
Notes on finalizers

Feature with significant performance impact
  - potentially large numbers of invocations:
    - array elements, list members
  - finalizer invoked twice in assignments with a function value as RHS

Finalizers of types with pointer components:
  - may need to consider reference counting to avoid undefined pointers

Non-allocatable variables in main program
  - have the implicit SAVE attribute → are not finalized

Finalizers will be covered more extensively in the advanced course
Input and Output to external storage
Terminology: Record and File

- **(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)
Handling I/O from Fortran

- **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:
  ```fortran
  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

```fortran
integer :: iunit
! define iunit
open(unit=iunit, &
    action='READ', &
    file='my_file', &
    form='FORMATTED', &
    status='OLD')
read(iunit, ...) ...
```

A unit may only be associated with **one file at a time**
- and vice versa
- close the file to disassociate

```fortran
! ... continued from before
close(unit=iunit)
open(iunit, &
    action='WRITE',
    file='new_file', &
    form='FORMATTED')
write(iunit, ...) ...
```

I/O control specification

- **I/O list**
- need not be in same program unit as **close**

- a write statement is not permissible here

- a read statement is not permissible here

will be detailed later ...
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**:

  ```fortran
  logical :: unit_exists, &
             unit_used
  iunit = ... some non-negative integer
  inquire(iunit, &
           exist=unit_exists, &
           opened=unit_used)
  
  if unit_exists is set to .true. and unit_used is set to .false.,
  an open on iunit will succeed.
  ```

- **Improved method**:
  - use the **newunit** specifier in open:

  ```fortran
  integer :: iunit
  open(newunit=iunit, ..., &
       file='myfile', ...)
  ```

  - this will define iunit with a (negative) integer that is connected to the specified file.

- **Note:**
  - Shell/OS limit on number of filehandles – not a Fortran issue
Specification of I/O lists

- **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:
    - 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()**

```fortran
  write(iu, *) a(1:3)  
  write(iu, *) a(1),a(2),a(3)  
  write(iu, *) (a(i),i=1,3)  
```

the three statements are equivalent
List-directed I/O

- A statement of the form

  ```fortran
  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**

- **Note:** slash in input field terminates I/O statement.

```fortran
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**

- contents of **my_file**

  ```
  5 .3 , , -1.2E-1, 1.4
  2*3.6 "No guarantee" .T.
  ```

  - quotes prevent unwanted splitting
  - blanks and/or comma are value separators

- variable z(1:6) after I/O statement

  ```
  .3 U -12 1.4 3.6 3.6
  ```

  - null value in input field
  - repeat factor

- reading „z“ would fail

- Note: slash in input field terminates I/O statement.
Edit descriptors

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)
Character string editing

- **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

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>type of list item</th>
<th>specific function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>character</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>integer</td>
<td>conversion to/from binary</td>
</tr>
<tr>
<td>I</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>integer</td>
<td>conversion to/from octal</td>
</tr>
<tr>
<td>Z</td>
<td>integer</td>
<td>conversion to/from hexadecimal</td>
</tr>
<tr>
<td>D</td>
<td>real</td>
<td>indicate extended precision and exponent</td>
</tr>
<tr>
<td>E</td>
<td>real</td>
<td>indicate exponent</td>
</tr>
<tr>
<td>EN</td>
<td>real</td>
<td>engineering notation</td>
</tr>
<tr>
<td>ES</td>
<td>real</td>
<td>scientific notation</td>
</tr>
<tr>
<td>F</td>
<td>real</td>
<td>fixed point (mostly …)</td>
</tr>
<tr>
<td>L</td>
<td>logical</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>any intrinsic</td>
<td>general editing: „auto-detection“ of edit descriptor to use</td>
</tr>
<tr>
<td>DT</td>
<td>derived type</td>
<td>user-defined „object-oriented“ I/O (aka UDDTIO)</td>
</tr>
</tbody>
</table>

Items marked green will be explicitly mentioned
# Table of control edit descriptors

<table>
<thead>
<tr>
<th>Descriptor(s)</th>
<th>function</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>BN, BZ</td>
<td>handling of embedded blanks in input fields</td>
<td>ignore / insert zero</td>
</tr>
<tr>
<td>SS, SP, S</td>
<td>output of leading signs</td>
<td>suppress/enforce/processor-defined</td>
</tr>
<tr>
<td>kp</td>
<td>scale numbers on input (or output)</td>
<td>usually by factor $10^{-k}$ (or $10^k$), except for scientific representation</td>
</tr>
<tr>
<td>Tn, (TRn nX), TLn</td>
<td>tabulation inside a record</td>
<td>move to position / right / left ($n$ in units of characters)</td>
</tr>
<tr>
<td>/</td>
<td>generate a new record</td>
<td>„linefeed“</td>
</tr>
<tr>
<td>: colon</td>
<td>terminate format control</td>
<td>when running out of I/O items</td>
</tr>
<tr>
<td>RU, RD, RZ, RN, RC, RP</td>
<td>change rounding mode for connection</td>
<td>up, down, to zero, to nearest, compatible, processor-defined</td>
</tr>
</tbody>
</table>

Items marked green will be explicitly mentioned
Formatted I/O: Format definition

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:

```fortran
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
```

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)

Field width and repeat factor

Field width is 10 (includes sign) width 2 – blank padding if not all characters needed

Bracketing and tabulation

Repeat count applies to parenthesised expression

control edit descriptor for right tabulation inserts a single blank

© LRZ 2009-18
Introduction to the Fortran programming language

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

Output will be

b0.233E02-0.715E00b7b9-3de

Bracketing and tabulation

write(iu, '(2(F5.2,1X,I2))') x(1),i(1),x(2),i(2)

Output will be

23.30bb7-0.72bb9
Using data and control edit descriptors (2)

```fortran
integer, allocatable :: csv_list(:)
allocate(csv_list(5))
csv_list(:) = [ 1, 2, 3, 4, 5 ]
```

### Unlimited repeat count and colon editing

```fortran
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

```fortran
write(iu, '(*(3I2,/)')) csv_list
```

Output:

```
b1 b2 b3 b4 b5
```
Undefined situations ...

```
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
```

**Input variables undefined**

```
Input File contains:
12345
-1.532E+102
b1.6732E+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

Output File contains:

```
***
********
-.1532+103
*******
```
... and how to avoid them

- **On output**
  - width ≥ 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

```fortran
i = 12345
write(iu,'(i0)') i
x = -1.532E102
write(iu,'(e11.4e3)') x
x = 1.6732E7
write(iu,'(f0.1)') x
```

- **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) → not dealt with in this course
  - for strings, the length parameter determines how many characters are read if the 'A' format is used

File contains:
```
12345
-.1532E+103
16732000.0
```
### Format exhaustion and reversion

**Assumption:**
- format string without components in parentheses
- more items in I/O list than edit descriptors are available

**Input:**
- format exhaustion → remainder of record is **skipped**
- otherwise similar to output
- example: file with contents

**Output:**
- will produce three records (the last one incomplete)
- format specification is repeated

```fortran
integer :: i(24)
i = ...
write(iu, '(10i4)') i
```

which is processed using

```fortran
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

```plaintext
... fmt='(i4, 3(2i3,2e10.3))' ...
... fmt='(i4, (2i3))' ...
```

penultimate right parenthesis in format string
Unformatted I/O

- Perform I/O without conversion to character strings
  - avoid conversion overhead
  - avoid possible roundoff errors
  - binary representation more space efficient

- Requires suitable OPEN specification:

  ```fortran
  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

  ```fortran
  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)
Unformatted I/O – portability issues

- Disadvantage: binary files may be unportable
  - padding
  - big- vs. little endian
  - large file treatment

- 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

Following now: 10th exercise session
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

<table>
<thead>
<tr>
<th>mode keyword</th>
<th>argument values (defaults in <strong>bold</strong>)</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>access=</td>
<td>'direct', '<strong>sequential</strong>' or 'stream'</td>
<td>determines access method</td>
</tr>
<tr>
<td>action=</td>
<td>'read', 'write' or 'readwrite'</td>
<td>determines I/O direction; default is processor-dependent.</td>
</tr>
<tr>
<td>asynchronous=</td>
<td>'yes' or 'no'</td>
<td>necessary (but not sufficient) for AIO</td>
</tr>
<tr>
<td>encoding=</td>
<td>'default' or 'utf-8'</td>
<td>UNICODE might work ...</td>
</tr>
<tr>
<td>form=</td>
<td>'formatted' or 'unformatted'</td>
<td>conversion method; default depends on access method.</td>
</tr>
<tr>
<td>position=</td>
<td>'asis', 'rewind' or 'append'</td>
<td>specifies the initial position of the file (sequential or stream access)</td>
</tr>
<tr>
<td>recl=</td>
<td>positive integer value</td>
<td>record length (in file storage units – often 1 byte) for direct or sequential access files</td>
</tr>
<tr>
<td>status=</td>
<td>'old', 'new', '<strong>unknown</strong>', 'replace' or 'scratch'</td>
<td>enforce condition on existence state of file before the OPEN statement is executed.</td>
</tr>
</tbody>
</table>
Changeable connection modes

- **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:**

  ```fortran
  real :: r = 2.33444e+2
  open(iu, ...)          ! assumption: open does not specify **sign**=
  write(iu, '(e11.4)') r
  write(iu, **sign='plus'**, '(e11.4)') r
  write(iu, **sign='suppress'**, '(e11.4)') r
  
  ! expected output
  0.2334E+03
  +0.2334E+03
  b0.2334E+03
  
  ! (first line is processor dependent)
  ```
# Table of changeable connection modes

<table>
<thead>
<tr>
<th>mode keyword</th>
<th>argument values (defaults in <strong>bold</strong>)</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank=</td>
<td>'null' or 'zero'</td>
<td>determine how blanks in input field are interpreted</td>
</tr>
<tr>
<td>decimal=</td>
<td>'comma' or '<strong>point</strong>'</td>
<td>set character used as decimal point during numeric conversion</td>
</tr>
<tr>
<td>delim=</td>
<td>'apostrophe', 'quote' or '<strong>none</strong>'</td>
<td>sets delimiter for character values in list-directed and namelist output</td>
</tr>
<tr>
<td>pad=</td>
<td>'yes' or 'no'</td>
<td>padding with blanks during input if more characters must be processed than contained in record</td>
</tr>
<tr>
<td>round=</td>
<td>'up', 'down', 'zero', 'nearest', '<strong>compatible</strong>', 'processor_defined'</td>
<td>set rounding mode for formatted I/O processing</td>
</tr>
<tr>
<td>sign=</td>
<td>'plus', 'suppress' 'processor_defined'</td>
<td>controls whether an optional plus sign will appear in formatted output</td>
</tr>
</tbody>
</table>
The CLOSE statement in more detail

- **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'

- **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
The INQUIRE statement

- 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
Examples for use of the INQUIRE statement

**Inquiry on unit**

```fortran
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**

```fortran
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

<table>
<thead>
<tr>
<th>mode keyword</th>
<th>argument variable type (and possible return values)</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct=, sequential=, stream=</td>
<td>character string: 'YES', 'NO', or 'UNKNOWN'</td>
<td>determine whether specified access is allowed for file</td>
</tr>
<tr>
<td>exist=</td>
<td>logical</td>
<td>determine whether a file or unit exists</td>
</tr>
<tr>
<td>formatted=, unformatted=</td>
<td>character string: 'YES', 'NO', or 'UNKNOWN'</td>
<td>determine whether (un)formatted I/O is allowed</td>
</tr>
<tr>
<td>name=</td>
<td>character string</td>
<td>find the name of a file connected to a unit</td>
</tr>
<tr>
<td>named=</td>
<td>logical</td>
<td>find out if file has a name</td>
</tr>
<tr>
<td>nextrec=</td>
<td>integer</td>
<td>find the next record number of a direct access file</td>
</tr>
<tr>
<td>number=</td>
<td>integer</td>
<td>identify unit connected to a file (-1 if no unit is connected)</td>
</tr>
<tr>
<td>opened=</td>
<td>logical</td>
<td>determine whether file or unit is connected</td>
</tr>
<tr>
<td>read=, write=, readwrite=</td>
<td>character string: 'YES', 'NO', or 'UNKNOWN'</td>
<td>determine whether named access mode is allowed for file</td>
</tr>
<tr>
<td>size=</td>
<td>integer</td>
<td>determine size of a file (in file storage units; -1 if the size cannot be determined)</td>
</tr>
</tbody>
</table>
Specifiers for data transfer statements

- 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
  - cannot be dealt with in this course
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 length `recl`
- Direct access file

```
record

rec = 1
2
3
```

- 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
  
  ```fortran
  integer(kind=lk) :: 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 2: Create direct access file**
- specify the maximum expected record length
  
  ```fortran
  open(unit=iu, file='da_file', &
       access='direct', &
       recl=max_length, &
       action='write', &
       status='replace')
  ```

**Step 3: Write a record**
- record not filled → remainder is undefined
  
  ```fortran
  do nr=...
    : ! set up x, y
    write(unit=iu, rec=nr) size(x), size(y), x, y
  end do
  ```

**Step 4: close file**
- usually, a single record
Open an existing direct access file for reading

\begin{verbatim}
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
\end{verbatim}

- 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)
Direct access files (4)

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
Concept of file position

- 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

![Diagram showing file position changes](image)
File positioning statements

- **BACKSPACE statement**
  - change file position to before the current record (if there is one), or else to before the previous record

- **ENDFILE statement**
  - write an EOF as the next record and position the file connection there

- **REWIND statement**
  - change position of file connection to initial position
  - allows to revert from undefined to defined file position

  ```fortran
  backspace(⟨unit⟩)
  ENDFILE(⟨unit⟩)
  rewind(⟨unit⟩)
  ```

  *Typically used*
  - for sequentially accessed files

  *beware*
  - performance issues
Error handling for I/O

- An I/O statement may fail
- Examples:
  - opening a non-existing file with status='OLD'
  - reading beyond the end of a file
  - 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

\[
\begin{align*}
\text{is_iostat_end}(i) \\
\text{is_iostat_eor}(i)
\end{align*}
\]

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

```fortran
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
```

Gracefully dealing with an EOF condition

```fortran
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
```
Namelist processing (1)

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

Example file:

```
file my_nml.dat

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

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

Required specifications

```
real :: flour, breadcrumbs, &
       salt, pepper
integer :: apples, pears
namelist /groceries/ flour, &
     breadcrumbs, salt, pepper
namelist / fruit / pears, apples
```

Reading the namelist

```
open(12, file='my_nml.dat', &
    form='formatted', action='read')
read(12, nml=groceries)
! pepper is undefined
read(12, nml=fruit)
```

- **NML** specifier instead of **FMT**
- multiple namelists require **same order** of reading as specified in file
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
**Internal I/O (1)**

- **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

```fortran
character(len=range(1)+1) :: i_char
integer :: i
: ! define i
write(i_char, fmt='(i0)') i
write(*, fmt='(a)') &
  trim(i_char)
```

- **Rules:**
  - no explicit connection needed
  - only formatted sequential access is possible
  - explicit, list-directed and namelist formatting is possible
Internal I/O (2)

- **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

```
character(len=...) :: my_fmt
integer, allocatable :: iarr(:)
integer :: iw

iw = ... ! prospective width e.g., 4
write(my_fmt, fmt= &
  '('''',i0,''',''i'',i0,''')''')' &
  ) size(iarr), iw
write(unit=..., fmt=my_fmt) iarr
```

value of my_fmt is '7i4'

Following now: 11th exercise session
This concludes the workshop

Thanks for your attention!