Fortran 90 for Beginners

Tadzio Hoffmann & Joachim Puls
summer semester 2010
<table>
<thead>
<tr>
<th></th>
<th>Literature, internet resources and compiler documentation</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Literature</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Internet resources</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Compiler documentation</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Fortran Syntax</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Data types</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Expressions</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Loops</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Decisions</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>Input/Output</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Arrays</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>Subroutines and functions</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>Modules</td>
<td>19</td>
</tr>
</tbody>
</table>
1 Literature, internet resources and compiler documentation

1.1 Literature

- Reference manuals
  - ‘Fortran 90’, RRZN (available at the LRZ).

- Textbooks

1.2 Internet resources

- Online-Tutorial at Univ. Liverpool
  http://www.liv.ac.uk/HPC/HTMLFrontPageF90.html

- Various resources
  - German Fortran Website
    http://www.fortran.de
  - Metcalf’s Fortran Information
    http://www.fortran.com/metcalf
  - Michel Olagnon’s Fortran 90 List
    http://www.fortran-2000.com/MichelList

1.3 Compiler documentation

- Documentation of installed compiler (man ifort or detailed in, e.g., /usr/share/modules/cmplrs/fortran_9.1.039/doc).

- Reference manuals by compiler vendors (on the web, e.g., by Cray/SGI, Sun, DEC/Compaq/HP, Intel).
2 Fortran Syntax

- line-oriented
- !: comment until end of line.
- statement separator/terminator: end of line or ;
  - example:
    
    ```fortran
    if(a>5) then; b=7; else; b=8; endif
    ```
    corresponds to
    ```fortran
    if(a>5) then
    b=7
    else
    b=8
    endif
    ```
- Maximum length of line: 132 characters; can be continued with &
  - example:
    ```fortran
    a=3*b + &
    7*c
    ```
- Identifiers (names of variables): up to 31 characters, consisting of A ... Z, 0 ... 9, _, have to begin with a letter. No difference between upper and lower case.
  - example: Abc_1 and aBc_1 are equal, but differ from Abc_2.
- Declaration of variables before executable statements.
  - Use always IMPLICIT NONE! In this way one is forced to declare all variables explicitly, and a lot of problems can be avoided. A missing implicit none-statement is equivalent to implicit integer (i-n), real (a-h,o-z) i.e., variables with names beginning with i to n will be integers, the others real.
    - example:
      ```fortran
      k=1.380662e-23
      ```
      yields \( k=0 \) (integer!) if \( k \) has not been explicitly declared as real.
- All programs, subroutines and functions must be ended (last line, except for comments) with
  ```fortran
  end
  ```
- Programs can (but do not have to) begin with program name, where name should be a useful name for the program.
  - example:
    ```fortran
    program test
    ```
3 Data types

- “elementary” data types: integer, real, complex, character, logical.
- “derived” types:
  - example:
    ```fortran
    type person
        character (len=20) :: name
        integer :: age
    end type
    type(person) :: myself
    myself%age=17
    ```
- attributes:
  - important for beginners
    dimension, allocatable, parameter, intent, kind, len
  - less important for beginners
    save, pointer, public, private, optional
  - Very useful (e.g., for the declaration of array dimensions): parameter
    Value already defined at compile-time, cannot be changed during run of program.
    Example:
    ```fortran
    integer, parameter :: np=3
    real, dimension(np) :: b ! vector of length 3
    real, dimension(np,np) :: x ! 3x3-matrix
    integer :: i
    
    do i=1,np
        b(i)=sqrt(i)
    enddo
    ```
- Different “kinds” of types: → “kind numbers” (e.g., different precision or representable size of numbers)
  - Warning!!! The resulting kind numbers can be different for different compilers and machines. Never use these numbers themselves, but assign them as a parameter!
  - Very useful! If all variables and constants have been declared by a “kind”-parameter, one single change (of this parameter) is sufficient to change the complete precision of the program.
  - Intrinsic functions:
    ```fortran
    selected_real_kind(mantissa_digits, exponent_range)
    selected_int_kind(digits)
    ```
  - If chosen precision is not available, these functions result in a negative value.
Example for correct use:

```fortran
integer, parameter :: sp = selected_real_kind(6,37)
or
integer, parameter :: sp = kind(1.)
integer, parameter :: dp = selected_real_kind(15,307)
or
integer, parameter :: dp = kind(1.d0)
integer, parameter :: qp = selected_real_kind(33,4931)
integer, parameter :: i4 = selected_int_kind(9)
integer, parameter :: i8 = selected_int_kind(16)
real (kind=sp) :: x,y ! or: real (sp) :: x,y
real (kind=dp) :: a,b ! ("double precision")
```

- Constants have type and kind as well:

  Examples:

  ```fortran
  integer: 1, 7890, 1_i8
  real: 1., 1.0, 1.e7, 1.23e-8, 4.356d-15, 1._dp, 2.7e11_sp
  complex: (0.,-1.), (2e-3,77._dp)
  character: 'Hello','I’m a character constant',
  ’xx’’yy’ → xx’yy
  "xx’yy" → xx’yy
  logical: .true., .false.
  “derived”: person("Meier",27)
  ```
4 Expressions

- numerical:
  - operators:
    - + sum
    - - difference
    - * product
    - / quotient
  - ** power
  - important intrinsic functions: \(\sin\), \(\cos\), \(\tan\), \(\arctan\), \(\exp\), \(\log\) (natural logarithm), \(\log_{10}\) (logarithm to base 10), \(\sqrt{}\), ...

Numerical operations are executed corresponding to the precision of the operand with higher precision:

- examples:
  - \(1/2 \rightarrow 0\)
  - \(1./2 \rightarrow 0.500000\)
  - \(1/2. \rightarrow 0.500000\)
  - \(1/2._\text{dp} \rightarrow 0.500000000000000\)
  - \(1+(1.,3) \rightarrow (2.000000,3.000000)\)

- logical:
  - operators:
    - \(\&\&\) boolean “and”
    - \(||\) boolean “or”
    - \(!\) boolean “not”
    - \(==\) or \(\equiv\) “equal”
    - \(!=\) or \(!\equiv\) “not equal”
    - \(>\) or \(\ge\) “greater than”
    - \(>=\) or \(\ge\) “greater than or equal”
    - \(<\) or \(\le\) “lower than”
    - \(<=\) or \(\le\) “lower than or equal”
  - intrinsic functions:
    - \(\llt\), \(\lle\), \(\lgt\), \(\lge\) comparison of characters (“lexically . . .”)

- character:
  - operators:
    - // concatenation
  - intrinsic functions: \(\text{char}, \text{ichar}, \text{trim}, \text{len}\)
5  LOOPS

Simple examples:

• "do"-loop (increment is optional, default = 1)

```fortran
  do i=1,10,2 ! begin, end, increment
     write(*,*) i,i**2
  enddo
```

Note: enddo and end do are equal.

```fortran
  do i=10,1 ! not executed
     write(*,*) i,i**2
  enddo
```

BUT

```fortran
  do i=10,1,-1 ! executed
     write(*,*) i,i**2
  enddo
```

if begin > end, increment MUST be present, otherwise no execution of loop

• "while"-loop

```fortran
  x= .2
  do while(x.lt..95)
     x=3.8*x*(1.-x)
     write(*,*) x
  enddo
```

• "infinite" loop

```fortran
  do ! "do forever". Exit required.
    write(*,*) 'Enter a number'
    read(*,*) x
    if(x.lt.0.) exit
    write(*,*) 'The square root of ',x,' is ',sqrt(x)
  enddo
```

• implied do-loop

```fortran
  write(*,*) (i,i**2,i=1,100)
```

Compare the following loops (identical results!)

```fortran
  do i=1,10,2
     write(*,*) i,i**2
  enddo
```
5. LOOPS

\[
i = 1 \\
do \quad \text{if}(i > 10) \quad \text{exit} \quad \text{write}(*,*) \ i, i^2 \quad i = i + 2 \\
enddo
\]

**Exit:** terminates loop (may also be named, in analogy to the “cycle” example below).

real, dimension(327) :: a ! instead of 327, better use an integer parameter here and in the following
integer :: i

! ... ! ... some calculations to fill vector a with numbers of increasing value ...
! ...
! search loop: searches for first number which is larger than 1.2345
do i = 1, 327
   if(a(i) > 1.2345) \text{exit}
enddo

! Note: value of counter after regular termination of loop
if(i eq 327+1) then
   write(*,*) 'index not found' stop
else
   write(*,*) 'index', i, ': value =', a(i)
endif

**Cycle:** starts new cycle of loop (may be named)

real, dimension(5,5) :: a
integer :: i, j
call random_number(a)
do i = 1, 5
   write(*,*) (a(i,j), j=1,5)
enddo

outer: do i = 1, 5 ! all matrix rows
   inner: do j = 1, 5 ! matrix columns, search loop:
      if(a(i,j) gt 0.8) then
         write(*,*) 'row', i, ': column', j, ',': a(i,j)
cycle outer
      endif
   enddo inner
   write(*,*) 'row ', i, ': nothing found'
endo outer

Note: if do loop is named, the \texttt{endo} statement \textit{must} be named as well.
6 Decisions

- Single-statement “If”
  
  \[ \text{if}(x \gt 0.) \ x = \sqrt{x} \]

- “Block If”:
  
  \[
  \begin{align*}
  \text{if}(x \gt 0.) \ & \ x = \sqrt{x} \\
  & \ y = y - x \\
  \text{endif}
  \end{align*}
  \]

  Note: \text{endif} and \text{end if} are equal.

- “If-Then-Else”:
  
  \[
  \begin{align*}
  \text{if}(x \lt 0.) \ & \ \text{write}(*,*) \ 'x \text{ is negative}' \\
  \text{else} & \\
  \text{if}(x \gt 0.) \ & \ \text{write}(*,*) \ 'x \text{ is positive}' \\
  \text{else} & \\
  \text{write}(*,*) \ 'x \text{ must be zero}' \\
  \text{endif} & \\
  \text{endif}
  \end{align*}
  \]

- “If-Then-Elseif- . . . -Else-Endif”: (cf. example above)
  
  \[
  \begin{align*}
  \text{if}(x \lt 0.) \ & \ \text{write}(*,*) \ 'x \text{ is negative}' \\
  \text{elseif}(x \gt 0.) \ & \ \text{write}(*,*) \ 'x \text{ is positive}' \\
  \text{else} & \\
  \text{write}(*,*) \ 'x \text{ must be zero}' \\
  \text{endif}
  \end{align*}
  \]

  Note: \text{elseif} and \text{else if} are equal.

- “Case”: (works only with integer, logical, character)
  
  \[
  \begin{align*}
  \text{read}(*,*) \ i \\
  \text{select case}(i) \\
  \text{case}(1) & \\
  \ & \ \text{write}(*,*) \ 'excellent' \\
  \text{case}(2,3) & \\
  \ & \ \text{write}(*,*) \ 'OK' \\
  \text{case}(4:6) & \\
  \ & \ \text{write}(*,*) \ 'shame on you' \\
  \text{case default} & \\
  \text{write}(*,*) \ 'impossible' \\
  \text{end select}
  \end{align*}
  \]
7 Input/Output

Most simple input/output statements (from/to terminal)

```fortran
real :: a
print*, 'Enter a real number'
read*, a
print*, 'input was ', a
```

Note the syntax (comma!) of the print*, read* statement, compared to the more general write, read statement considered from now on.

```fortran
write(*,*) means write(unit=*,fmt=*)
```

- Units:
  ```fortran
  open(1,file='output')
  write(1,*) 'Hello, world!'
  close(1)
  ```

- Error handling (end=n, err=m)
  ```fortran
  program read
  implicit none
  integer, parameter :: m=10
  integer :: i
  real, dimension (m) :: a
  real :: t

  open (77,file='numbers')
  i=0
  do
    read(77,*,end=200,err=100) t
    i=i+1
    if(i.gt.m) then
      write(*,*) 'array too small.', &
      ' increase m and recompile.'
      close(77)
      stop
    endif
    a(i)=t
  enddo

  100 continue
  write(*,*) 'read error in line',i+1
  close(77)
  stop
  200 continue
  write(*,*) i,' numbers read'
  close(77)
  write(*,*) a(1:i)
  end
  ```
• Input/output into character-variable ("internal file")

```fortran
character (len=20) :: a
write(a,*), "Hello, world!"
```

• Formatted input/output

Note: explicitly formatted input rather complex, use list-directed input instead (i.e., fmt=*) unless you are completely sure what you are doing!

```fortran
write(*,700) 1,1.23,(7.,8.),'Hello',.true.
write(*,701)
write(*,702)
write(*,'(i5,e12.4e3,2f8.2,1x,a3,l7)') 1,1.23,(7.,8.),'Hello',.true.
```

results in

```
1 0.1230E+001 7.00 8.00 Hel T
12345678901234567890123456789012345678901234567890
1 2 3 4 5
1 0.1230E+001 7.00 8.00 Hel T
```

• If end of format reached, but more items in input/output list: switch to next line, continue with corresponding format descriptor (in most cases, the first one).

```fortran
write(*,700) 1,1.23,(7.,8.),'Hello',.true.,3,4.
write(*,701)
write(*,702)
```

results in

```
1 0.1230E+001 7.00 8.00 Hel T
1 2 3 4 5
1 0.1230E+001 7.00 8.00 Hel T
```

• The format can be specified either by a separate statement (with label), or, more directly, by a character-constant or -variable. (Note: the outer parentheses are part of the format-specification)

```fortran
real :: x
character (len=8) :: a

write(*,123) x
write(*,'(es10.2)') x
write(*,a) x
```

...
• “Edit descriptors”:

<table>
<thead>
<tr>
<th>Format</th>
<th>Value to be written</th>
<th>Output (spaces indicated by “_”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i5</td>
<td>12</td>
<td>___12</td>
</tr>
<tr>
<td>i5.3</td>
<td>12</td>
<td>__012</td>
</tr>
<tr>
<td>i5.3</td>
<td>1234</td>
<td>__1234</td>
</tr>
<tr>
<td>i7.7</td>
<td>1234</td>
<td>0001234</td>
</tr>
<tr>
<td>i7.7</td>
<td>−1234</td>
<td>*******</td>
</tr>
<tr>
<td>i7.6</td>
<td>−1234</td>
<td>−001234</td>
</tr>
<tr>
<td>b16</td>
<td>1234</td>
<td>-----10011010010! binary</td>
</tr>
<tr>
<td>b16.14</td>
<td>1234</td>
<td>__00010011010010!</td>
</tr>
<tr>
<td>o8</td>
<td>1234</td>
<td>___2322</td>
</tr>
<tr>
<td>o8.8</td>
<td>1234</td>
<td>00002322</td>
</tr>
<tr>
<td>z6</td>
<td>1234</td>
<td>___4D2</td>
</tr>
<tr>
<td>z6.5</td>
<td>1234</td>
<td>_004D2</td>
</tr>
<tr>
<td>e12.4</td>
<td>−1234.</td>
<td>__-0.1234E+04!</td>
</tr>
<tr>
<td>e12.4</td>
<td>−1.234e12</td>
<td>__-0.1234E+13</td>
</tr>
<tr>
<td>e12.4</td>
<td>−1.234e12_dp</td>
<td>__-0.1234+124</td>
</tr>
<tr>
<td>e14.4e3</td>
<td>−1.234e12_dp_dp</td>
<td>__-0.1234E+124</td>
</tr>
<tr>
<td>f12.4</td>
<td>−1234.</td>
<td>__-1234.0000</td>
</tr>
<tr>
<td>f12.4</td>
<td>−1.234</td>
<td>_____-1.2340</td>
</tr>
<tr>
<td>f12.4</td>
<td>−1.234e12</td>
<td>***********! exponential</td>
</tr>
<tr>
<td>e14.4</td>
<td>−1.234e5</td>
<td>___-0.1234E+06!</td>
</tr>
<tr>
<td>e14.4</td>
<td>−1.234e5</td>
<td>___-1.2340E+05!</td>
</tr>
<tr>
<td>e14.4</td>
<td>−1.234e5</td>
<td>___-123.4000E+03!</td>
</tr>
<tr>
<td>a</td>
<td>'Hello, world!'</td>
<td>Hello,_world!</td>
</tr>
<tr>
<td>a8</td>
<td>'Hello, world!'</td>
<td>Hello,_w</td>
</tr>
<tr>
<td>a15</td>
<td>'Hello, world!'</td>
<td>__Hello,_world!</td>
</tr>
</tbody>
</table>

Examples:
8 Arrays

- Examples:

```plaintext
real, dimension(2,2) :: a        ! 2x2-matrix
real, dimension(3:4,-2:-1) :: q  ! 2x2-matrix
integer, parameter :: m=27, n=123
real, dimension(n,m) :: b,c       
real, dimension(m) :: x,y         
```

- Intrinsic functions: shape, size, lbound, ubound:

```plaintext
shape(b)    → 123, 27 (= n,m)
size(b)     → 3321 (= 123*27)
size(b,1)   → 123
size(b,2)   → 27
lbound(q,2) → -2
ubound(q,1) → 4
```

- Array-constructor (array-constant in program):
  
  - example:
    ```plaintext
    x=(/ 1.,2.,3.,4.,5. /) 
y=(/ (0.1*i, i=1,m) /) ! --> 0.1, 0.2, 0.3, 0.4, 0.5, ... 
    ```
    
    - Unfortunately, this works only for one-dimensional arrays. Construction of more-
      dimensional arrays with reshape:
      ```plaintext
      a=reshape( (/ 1.,2.,3.,4. /), (/ 2,2 /))
      ```

      Warning!!! Warning!!! Warning!!!
      ! Sequence of storage in Fortran!
      "first index runs fastest."
      ```plaintext
      a(1,1)=1., a(2,1)=2., a(1,2)=3., a(2,2)=4.
      ```
      
      - Array syntax: operations for complete array (element-wise) in one statement.
        
        - example:
          ```plaintext
          ! ... declaration of parameters n,m
          real, dimension(n,m) :: b,c
          b=sin(c)
          ```

          gives same result as
          ```plaintext
          real, dimension(n,m) :: b,c
          integer :: i,j
          do i=1,n
            do j=1,m
              b(i,j)=sin(c(i,j))
            enddo
          enddo
          ```

14
° operations on specific “array sections” possible as well:

```fortran
real, dimension(10) :: u, v
real, dimension(5,4) :: w
u(2:10:2) = sin(w(:,1)) ! shapes must match
v(1:3) = 5.
! alternatively: v(:3) = 5.
```

\( u(i:j:k) \) means: those elements of \( u \) starting with index \( i \) until index \( j \), only every \( k \)-th element. \( k \) is optional (default = 1), missing \( i \) or \( j \) implies lower or upper array boundary, respectively.

° “where”-block: allows for operation(s) on specific sections determined from the \textbf{where} condition. Very useful, e.g., to avoid divisions by zero:

```fortran
where(x.eq.0.)
  y=1.
elsewhere
  y=sin(x)/x
endwhere
```

• Difference of array-operations and DO-loop in case of recurrence. Array-operations evaluate total rhs first! Compare:

```fortran
do i=2,m
  x(i)=x(i)+x(i-1)
enddo
```

with

\[
x(2:m) = x(2:m) + x(1:m-1)
\]
9 Subroutines and functions

- Rationale:
  1. actions/operations which have to be performed more than once
  2. one big problem split into clearer and simpler sub-problems

- Example:
  
  ```fortran
  program main
    implicit none
    integer i
    real :: x,y,sinc
    do i=0,80,2
      x=i/10.
      y=sinc(x)
      write(*,*) x,y
    ! or print*,x,y
  enddo
  call output(0,80,2)
  end
  
  function sinc(x)
    implicit none
    real :: x,sinc
    if(x.eq.0.) then
      ! be careful with comparison to real numbers because of rounding errors
      ! better: if (abs(x).lt.1.e-16) then
      sinc=1.
    else
      sinc=sin(x)/x
    endif
  end
  
  subroutine output(a,e,s)
    integer, intent(in) :: a,e,s
    real :: x,y,sinc
    integer :: I
    open(1,file='sinc.data')
    do i=a,e,s
      x=i/10.
      y=sinc(x)
      write(1,10) x,y
    enddo
    close(1)
  10 format(2e14.6)
  end
  
  Disadvantage of this definition of sinc: cannot be called with array arguments, as is, e.g.,
  the case for the intrinsic sin function (improved definition on page 22).
• Passing of arrays to subroutines/functions; local arrays
  ◦ Who reserves storage for arrays? Calling or called routine?
  ◦ Size of array known at compile time or only at run time?

Several possibilities

```fortran
program main
  implicit none
  ! ...
  integer, parameter :: n=100
  real, dimension(n) :: a,b,c,d
  call sub(a,b,c,d,n)
end

subroutine sub(u,v,w,x,m)
  real, dimension(100) :: u ! constant size
  real, dimension(m) :: v ! adjustable size
  real, dimension(*) :: w ! assumed size
  real, dimension(:,*) :: x ! assumed shape (needs interface block in calling routine!)
  real, dimension(100) :: y ! constant size (local)
  real, dimension(m) :: z ! automatic (local)
  real, dimension(:,), allocatable :: t ! deferred-shape (local)
  !...
  allocate(t(m))
  !...
  write(*,*) u,v,x,y,z,t ! assumed size needs explicit indexing
  write(*,*) w(1:m) ! because upper bound is unknown
  !...
  deallocate(t)
end
```

◦ Recommendation: use adjustable size or assumed shape; avoid assumed size.

◦ Note: maximum size of automatic arrays system dependent. To be on the safe side, use only “small” automatic arrays if necessary. Otherwise, use constant size with sufficient dimension.
• transfer of “array sections” (special case of “assumed shape”), requires interface block, see also page 21.

    program main
    implicit none
    interface
       subroutine sub(x)
          real, dimension(:) :: x
       end subroutine
    end interface
    integer, parameter :: n=100
    real, dimension(n) :: a
    call sub(a(1:50:3))
    end

    subroutine sub(x)
       real, dimension(:) :: x
       write(*,*) shape(x)
    end

Note: interface blocks should be collected in a specific module (see next section).
10 Modules

- Usage/rationale:
  1. Declaration of subroutines/functions and interface blocks. Calling program needs only to “use module-name”.
  2. “Global” variables: can be used by different routines without explicit passing as arguments.
     Recommendation: use module-name, only: var1, var2 ...
  3. Encapsulation of functions and corresponding variables.

- Precision control: definition of kind-numbers

```fortran
module my_type
  integer, parameter :: ib = selected_int_kind(9) !integer*4
  integer, parameter :: sp = selected_real_kind(6,37) !real*4 or sp = kind(1.)
  ! integer, parameter :: sp = selected_real_kind(15,307) !real*8 or dp = kind(1.d0)
  ! Useful trick: precision of following routines can be easily changed
  ! from single to double precision by alternatively
  ! commenting/uncommenting the statements defining sp
end module my_type
```

```fortran
program random
  use my_type ! use statement(s) must be given before further declarations
  implicit none

  integer(ib) :: i
  real(sp) :: x

  do i=1,5
    call random_number(x)
    print*,x
  enddo

end
```
Example for use of “global” variables without explicit argument passing:

```fortran
module common
    implicit none
    real :: x,y=5.
end module

program test
    implicit none
    call sub1
    call sub2
    call sub3
end

subroutine sub1
    use common, only: x
    implicit none
    real :: y
    x=3.
y=1.
    write(*,*) x,y
end

subroutine sub2
    use common, only: x
    implicit none
    write(*,*) x
    x=7.
end

subroutine sub3
    use common
    implicit none
    write(*,*) x, y
end
```
• Declaration of subroutine(s) or corresponding interfaces in module (see example page 18)

```fortran
module mymod
! no explicit interface block if routine is "contained"
contains
subroutine mysub(x)
  implicit none
  real, dimension(:) :: x
  write(*,*) shape(x)
end subroutine
end module

program main
use mymod
implicit none
integer, parameter :: n=100
real, dimension(n) :: a
call mysub(a(1:50:3))
end

or recommended way to proceed

module mymod
! interface block necessary if routine is defined elsewhere
interface
  subroutine mysub(x)
    implicit none
    real, dimension(:) :: x
  end subroutine
end interface
end module

program main
use mymod
implicit none
integer, parameter :: n=100
real, dimension(n) :: a
call mysub(a(1:50:3))
end

subroutine mysub(x)
  implicit none
  real, dimension(:) :: x
  write(*,*) shape(x)
end subroutine
```

21
• Example: functions with arguments being either scalars or arrays (cf. example page 16)

```fortran
module sincm
  interface sinc
    module procedure sinca, sincs
  end interface
  contains
    function sinca(x) result(z) ! array
      implicit none
      real, dimension(:) :: x
      real, dimension(size(x)) :: z
      where(x.eq.0.)
      z=1.
      elsewhere
      z=sin(x)/x
      endwhere
    end function
    function sincs(x) result(z) ! scalar
      implicit none
      real :: x,z
      if(x.eq.0.) then
        z=1.
      else
        z=sin(x)/x
      endif
    end function
end module

program main
use sincm
implicit none

integer, parameter :: m=100
real, dimension(m) :: x,y
integer :: i

x=(/ (0.2*i,i=1,m) /)
y=sinc(x) ! array sinc
write(*,777) (i,x(i),y(i),i=1,m)
777 format(i5,2e12.4)
write(*,*) sinc(1.23) ! scalar sinc
end
```

22