Ludwigs-Maximilians-Universität München
— Departement for Physics —

University Observatory

Fortran 90 for Beginners

Tadziu Hoffmann & Joachim Puls

summer semester 2010

Contents

1 Literature, internet resources and compiler documentation **3**
- 1.1 Literature . . . 3
- 1.2 Internet resources . . . 3
- 1.3 Compiler documentation . . . 3

2 Fortran Syntax **4**

3 Data types **5**

4 Expressions **7**

5 Loops **8**

6 Decisions **10**

7 Input/Output **11**

8 Arrays **14**

9 Subroutines and functions **16**

10 Modules **19**

1 Literature, internet resources and compiler documentation

1.1 Literature

- **Reference manuals**
  - **Gehrke, W.**, *Fortran90 Referenz-Handbuch*, 1991, Hanser, München, ISBN 3446163212
  - **'Fortran 90'**, RRZN (available at the LRZ).
- **Textbooks**
  - **Adams, J.C., et al.**: *Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*, 2008, Springer, Berlin, ISBN 1846283787
  - **Metcalf, M., et al.**: *Fortran 95/2003 explained*, 2004, Oxford Univ. Pr., ISBN 0198526938 (paperback)

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:

    ```
    if(a>5) then; b=7; else; b=8; endif
    ```

    corresponds to

    ```
    if(a>5) then
      b=7
    else
      b=8
    endif
    ```

- Maximum length of line: 132 characters; can be continued with `&`
  - example:

    ```
    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:

    ```
    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

  ```
  end
  ```

- Programs can (but do not have to) begin with `program name`, where `name` should be a useful name for the program.
  - example:

    ```
    program test
    ```

3 Data types

- "elementary" data types: `integer`, `real`, `complex`, `character`, `logical`.
- "derived" types:
  - example:

    ```
    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:

    ```
    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 paramenter!
  - 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:

    `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:

```
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:

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`, `atan`, `exp`, `log` (natural logarithm), `log10` (logarithm to base 10), `sqrt`, . . .

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

  - examples:

`1/2`	→	0
`1./2`	→	0.5000000
`1/2.`	→	0.5000000
`1/2._dp`	→	0.500000000000000
`1+(1.,3)`	→	(2.000000,3.000000)

- logical:
  - operators:

`.and.`	boolean "and"
`.or.`	boolean "or"
`.not.`	boolean "not"
`.eq.` or `==`	"equal"
`.ne.` or `/=`	"not equal"
`.gt.` or `>`	"greater than"
`.ge.` or `>=`	"greater than or equal"
`.lt.` or `<`	"lower than"
`.le.` or `<=`	"lower than or equal"

  - intrinsic functions:

`llt`, `lle`, `lgt`, `lge`	comparison of characters ("lexically . . .")

- character:
  - operators:

`//`	concatenation

  - intrinsic functions: `char`, `ichar`, `trim`, `len`

5 Loops

Simple examples:

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

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

  Note: `enddo` and `end do` are equal.

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

  BUT

  ```
  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

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

- "infinite" loop

  ```
  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

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

Compare the following loops (identical results!)

```
do i=1,10,2
  write(*,*) i,i**2
enddo
```

```
  i=1
  do
    if(i.gt.10) exit
    write(*,*) i,i**2
    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).gt.1.2345) 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:
                     ! searches for first number > 0.8 in row i
    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'
  enddo outer
```

Note: if do loop is named, the `enddo` statement *must* be named as well.

6 Decisions

- Single-statement "If"

```
if(x.gt.0.) x=sqrt(x)
```

- "Block If":

```
if(x.gt.0.) then
  x=sqrt(x)
  y=y-x
endif
```

Note: `endif` and `end if` are equal.

- "If-Then-Else":

```
if(x.lt.0.) then
  write(*,*) 'x is negative'
else
  if(x.gt.0.) then
    write(*,*) 'x is positive'
  else
    write(*,*) 'x must be zero'
  endif
endif
```

- "If-Then-Elseif- . . . -Else-Endif": (cf. example above)

```
if(x.lt.0.) then
  write(*,*) 'x is negative'
elseif(x.gt.0.) then
  write(*,*) 'x is positive'
else
  write(*,*) 'x must be zero'
endif
```

Note: `elseif` and `else if` are equal.

- "Case": (works only with `integer, logical, character`)

```
read(*,*) i
select case(i)
  case(1)
    write(*,*) 'excellent'
  case(2,3)
    write(*,*) 'OK'
  case(4:6)
    write(*,*) 'shame on you'
  case default
    write(*,*) 'impossible'
end select
```

7 Input/Output

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

```
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.

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

- Units:

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

- Error handling (`end=`n, `err=`m)

  ```
      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")

```
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!

```
    write(*,700) 1,1.23,(7.,8.),'Hello',.true.
    write(*,701)
    write(*,702)
700 format(i5,e12.4e3,2f8.2,1x,a3,l7)
701 format('12345678901234567890123456789012345678901234567890')
702 format('         1         2         3         4         5')
    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).

```
    write(*,700) 1,1.23,(7.,8.),'Hello',.true.,3,4.
700 format(i5,e12.4e3,2f8.2,1x,a3,l7)
```

results in

```
    1 0.1230E+001    7.00    8.00 Hel      T
    3 0.4000E+001
```

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

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

     ...

     write(*,123) x
 123 format(es10.2)

     write(*,'(es10.2)') x

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

- "Edit descriptors":

integer:	`i`, `b`, `o`, `z`		
real:	`d`, `e`, `f`, `g`, `es`, `en`		
character:	`a`		
logical:	`l`		
other:	n	(number)	repeat following descriptor n times
	`x`	space	
	`/`	new line	
	`'...'`	literal text	
	`(...)`	group	
	`p`	scale	

Examples:

format	value to be written	output (spaces indicated by "_")	
`i5`	12	`___12`	
`i5.3`	12	`__012`	
`i5.3`	1234	`_1234`	
`i7.7`	1234	`0001234`	
`i7.7`	−1234	`*******`	
`i7.6`	−1234	`-001234`	
`b16`	1234	`_____10011010010`	! binary
`b16.14`	1234	`__00010011010010`	
`o8`	1234	`____2322`	! octal
`o8.8`	1234	`00002322`	
`z6`	1234	`___4D2`	! hexadecimal
`z6.5`	1234	`_004D2`	
`e12.4`	−1234.	`_-0.1234E+04`	
`e12.4`	−1.234e12	`_-0.1234E+13`	
`e12.4`	−1.234e123_dp	`_-0.1234+124`	
`e14.4e3`	−1.234e123_dp	`__-0.1234E+124`	
`f12.4`	−1234.	`__-1234.0000`	
`f12.4`	−1.234	`_____-1.2340`	
`f12.4`	−1.234e12	`************`	
`e14.4`	−1.234e5	`___-0.1234E+06`	! exponential
`es14.4`	−1.234e5	`___-1.2340E+05`	! scientific
`en14.4`	−1.234e5	`_-123.4000E+03`	! engineering
`a`	'Hello, world!'	`Hello,_world!`	
`a8`	'Hello, world!'	`Hello,_w`	
`a15`	'Hello, world!'	`__Hello,_world!`	

8 Arrays

- Examples:

```
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`:

`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:

```
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`:

```
a=reshape( (/ 1.,2.,3.,4. /), (/ 2,2 /))
```

    Warning!!! Warning!!! Warning!!!
    ! Sequence of storage in Fortran!
    “first index runs fastest.”

    a(1,1)=1., a(2,1)=2., a(1,2)=3., a(2,2)=4.

    $\rightarrow \boxed{\begin{matrix} 1. & 3. \\ 2. & 4. \end{matrix}}$

- Array syntax: operations for *complete* array (element-wise) in *one statement.*
  - example:

```
! ... declaration of parameters n,m
real, dimension(n,m) :: b,c
b=sin(c)
```

    gives same result as

```
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
```

- operations on specific "array sections" possible as well:

```
  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 `where` condition. Very useful, e.g., to avoid divisions by zero:

```
  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

```
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:

```
  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

```
  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

```
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

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:

```
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)

```
  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
```

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

```
    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
```