554027 Modern Fortran Programming for Chemists and Physicists

Dr Pekka Manninen
manninen@cray.com

About this course

- **Lectures**: The course consists of 14 hours of face-to-face learning sessions. Lectures in Period III on Mondays from 2.15 pm to 4.00 pm (Jan 13 – Feb 24, 2014).
- **Room**: Computer classroom D211, Physicum building, Kumpula campus.
- **Credits**: 2 ECTS. Completing the programming assignments given after each lecture is required for the credits.
- **Literature**: Metcalf, Reid, Cohen: Modern Fortran Explained (Oxford University Press, 2011); Haataja, Rahola, Ruokolainen: Fortran 95/2003 4th ed. (CSC, 2007). Lecture notes and other materials will be available online.
  - Lectures have their origin on the numerous Fortran courses given at CSC by PM and other people (Sami Saarinen, Sami Ilvonen,...)
- **Course page:** http://www.chem.helsinki.fi/~manninen/fortran2014

Course outline (discussion)

Session	Topics
Jan 13	Basic syntax, program controls, structured programming
Jan 20	Modular programming; Fortran arrays
Jan 27	Input/output: formatting, writing/reading files
Feb 3	Derived datatypes, procedure interfaces, operator overloading
Feb 10	Procedure attributes, parameterized types, abstract interfaces, procedure pointers, interoperability with C language
Feb 17	Parallel programming with Fortran coarrays
Feb 24	Extended types, polymorphism, type-bound procedures

Web resources

- CSC's Fortran95/2003 Guide (in Finnish) for free
  http://www.csc.fi/csc/julkaisut/oppaat
- Fortran wiki: a resource hub for all aspects of Fortran programming
  http://fortranwiki.org
- GNU Fortran online documents
  http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gfortran
- Code examples
  http://www.nag.co.uk/nagware/examples.asp
  http://www.personal.psu.edu/jhm/f90/progref.html
  http://www.physics.unlv.edu/~pang/cp_f90.html

Lecture I: Getting started with Fortran

Outline

- First encounter with Fortran
- Variables and their assignment
- Control structures

Why learn Fortran?

- Well suited for numerical computations
  - Likely over 50% of scientific applications are written in Fortran
- Fast code (compilers can optimize well)
- Handy array data types
- Clarity of code
- Portability of code
- Optimized numerical libraries available

Fortran through the ages

- John W. Backus et al (1954): The IBM Mathematical **For**mula **Tran**slating System
- Early years development: Fortran II (1958), Fortran IV (1961), Fortran 66 & Basic Fortran (1966)
- Fortran 77 (1978)
- Fortran 90 (1991) major revision, Fortran 95 (1995) a minor revision to it

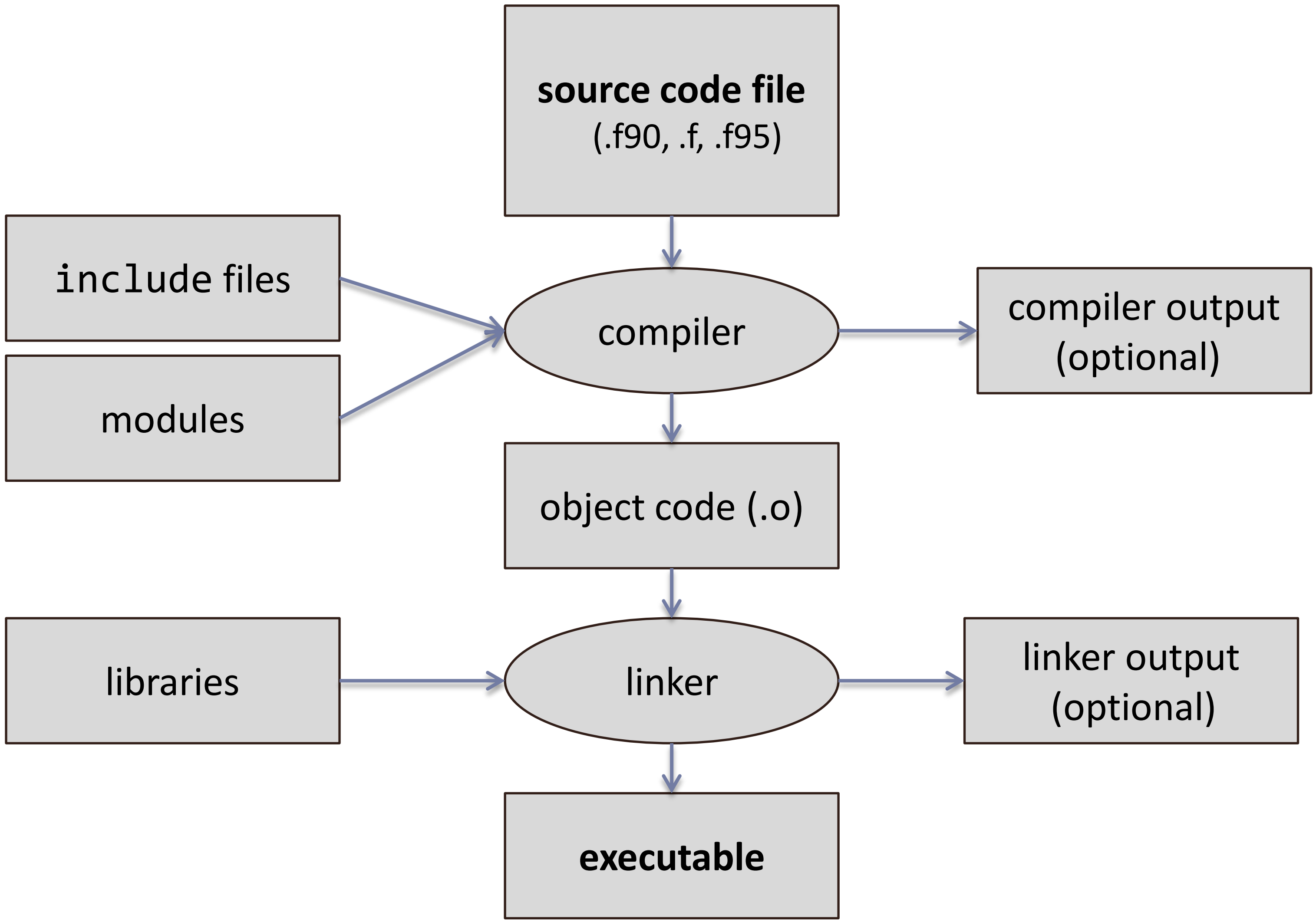
Fortran through the ages

- Fortran 2003: major revision, adding e.g. object-oriented features
  - ”Fortran 95/2003” is the current *de facto* standard
- The latest standard is Fortran 2008 (approved 2010), a minor upgrade to 2003
- All relevant compilers implement fully the 2003 standard
  - Fortran 2008 features still under construction, Cray and Intel compilers most complient

Look & Feel

```
program square_root_example
! comments start with an exclamation point.
! you will find data type declarations, couple arithmetic operations
! and an interface that will ask a value for these computations.
 implicit none
 real :: x, y
 intrinsic sqrt ! fortran standard provides many commonly used functions
 ! command line interface. ask a number and read it in
 write (*,*) 'give a value (number) for x:'
 read (*,*) x
 y=x**2+1   ! power function and addition arithmetic
 write (*,*) 'given value for x:', x
 write (*,*) 'computed value of x**2 + 1:', y
 ! print the square root of the argument y to screen
 write (*,*) 'computed value of sqrt(x**2 + 1):', sqrt(y)
end program square_root_example
```

Compiling and linking



Variables

```
implicit none
integer :: n0
real :: a, b
real :: r1
complex :: c
complex :: imag_number=(0.1, 1.0)
character(len=80) :: place
character(len=80) :: name='james bond'
logical :: test0 = .true.
logical :: test1 = .false.
real, parameter :: pi=3.14159
```

Variables must be *declared* at the beginning of the program or procedure

The *intrinsic* data types in Fortran are **INTEGER, REAL, COMPLEX, CHARACTER** and **LOGICAL**

They can also be given a value at declaration

Constants defined with the PARAMETER clause – they cannot be altered after their declaration

Assignment statements

```
program numbers
  implicit none
  integer :: i
  real :: r
  complex :: c, cc
  i = 7
  r = 1.618034
  c = 2.7182818   !same as c = cmplx(2.7182818)
  cc = r*(1,1)
  write (*,*) i, r, c, cc
end program
```

Automatic change of representation, works between all numeric intrinsic data types

Output (one integer and real and two complex values) :

```
7  1.618034  (2.718282, 0.000000)  (1.618034, 1.618034)
```

How can I convert numbers to character strings and vice versa? See "internal I/O" in the File I/O lecture.

Arrays

```
integer, parameter :: m = 100, n = 500
integer :: idx(m)
real :: vector(0:n-1)
real :: matrix(m, n)
character (len = 80) :: screen(24)

! or, equivalently,

integer, dimension(m) :: idx
real, dimension(0:n-1) :: vector
real, dimension(m, n) :: matrix
character(len=80), dimension(24) :: screen
```

By default, indexing starts from 1

Operators

Arithmetic

```
real :: x, y
integer :: i
x=2.0**(-i)  ! power function
x=x*real(i)  ! multiplication and type
               change
x=x/2.0      ! division
i=i+1        ! addition
i=i-1        ! subtraction
```

Relational

```
.lt. or <    ! less than
.le. or <=   ! less than or equal to
.eq. or ==   ! equal to
.ne. or /=   ! not equal to
.gt. or >    ! greater than
.ge. or >=   ! greater than or equal to
```

Logical operators

```
.not.      ! logical negation
.and.      ! logical conjunction
.or.       ! logical inclusive
             disjunction
```

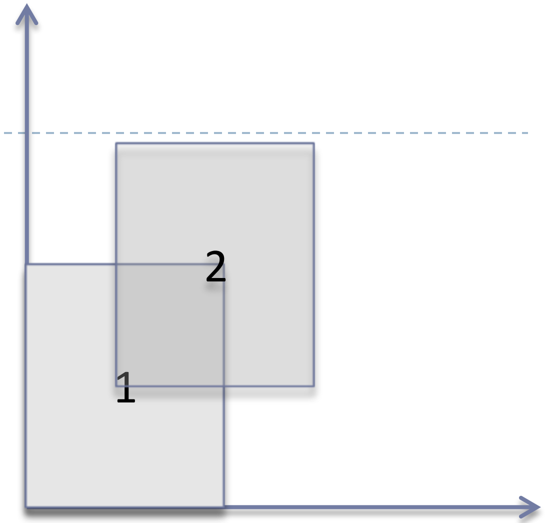
Control structures: conditionals

```
program test_if
 implicit none
 real :: x,y,eps,t

 write(*,*)' give x and y :'
 read(*,*) x, y

 if (abs(x) > 0.0) then
     t=y/x
 else
     write(*,*)'division by zero'
     t=0.0
 end if
 write(*,*)' y/x = ',t
end program
```

Conditionals example




```
program placetest
 implicit none
 logical :: in_square1, in_square2
 real :: x,y
 write(*,*) 'give point coordinates x and y'
 read (*,*) x, y
 in_square1 = (x >= 0. .and. x <= 2. .and. y >= 0. .and. y <= 2.)
 in_square2 = (x >= 1. .and. x <= 3. .and. y >= 1. .and. y <= 3.)
 if (in_square1 .and. in_square2) then       ! inside both
    write(*,*) ‘point within both squares’
 else if (in_square1) then                   ! inside square 1 only
    write(*,*) ‘point inside square 1’
 else if (in_square2) then                   ! inside square 2 only
    write(*,*) ‘point inside square 2’
 else                                        ! both are .false.
    write(*,*) ‘point outside both squares’
 end if
end program placetest
```

Control structures: loops

```
! loop with an integer counter (count controlled)
integer :: i, stepsize, numberofpoints
integer, parameter :: max_points=100000
real :: x_coodinate(max_points), x, totalsum
...
stepsize=2
do i = 1, max_points, stepsize
   x_coordinate(i) = i*stepsize*0.05
end do

! condition controlled loop
totalsum = 0.0
read(*,*) x
do while (x > 0)
   totalsum = totalsum + x
   read(*,*) x
end do
```

Control structures: loops

```
! do loop without loop control

real :: x, totalsum, eps
totalsum = 0.0
do
  read(*,*) x
  if (x < 0) then
     exit              ! exit the loop
  else if (x > upperlimit) then
     cycle             ! do not execute any statements but
                       ! cycle back to the beginning of the loop
  end if
  totalsum = totalsum + x
end do
```

Control structures example

```
program gcd
! computes the greatest common divisor, euclidean algorithm
  implicit none
  integer :: m, n, t
  write(*,*)' give positive integers m and n :'
  read(*,*) m, n
  write(*,*)'m:', m,' n:', n
  positive_check: if (m > 0 .and. n > 0) then
     main_algorithm: do while (n /= 0)
        t = mod(m,n)
        m = n
        n = t
     end do main_algorithm
     write(*,*)'greatest common divisor: ',m
   else
     write(*,*)'negative value entered'
   end if positive_check
end program gcd
```

Labels can be given to control structures and used in conjunction with e.g. exit and cycle statements

Control structures: select case

- SELECT CASE statements matches the entries of a list against the case index
  - Only one found match is allowed
  - Usually arguments are character strings or integers
  - DEFAULT branch if no match found

```
...
integer :: i
logical :: is_prime_number
...
select case (i)
  case (2,3,5,7)
    is_prime_number = .true.
  case (1,4,6,8:10)
    is_prime_number = .false.
  case default
    is_prime_number=test_prime_number(i)
end select
...
```

Source code remarks

- A variable name can be no longer than 31 characters
  - containing only letters, digits or underscore
  - must start with a letter
- Maximum row length is 132 characters
- There can be max 39 continuation lines
  - if a line is ended with ampersand (&), the line continues onto the next line
- No distinction between lower and uppercase characters
  - character strings are case sensitive

Source code remarks

```
! character strings are case sensitive
character(len=32) :: ch1, ch2
logical :: ans
ch1 = 'a'
ch2 = ‘A'
ans = ch1 .eq. ch2
write(*,*) ans     ! output from that write statement is: f
! when strings are compared
! the shorter string is extended with blanks
write(*,*) 'a' .eq. 'a '    !output: t
write(*,*) 'a' .eq. ' a'    !output: f
! statement separation: newline and semicolon, ;
! semicolon as a statement separator
a = a * b; c = d**a
! the above is equivalent to following two lines
a = a * b
c = d**a
```

Structured programming

- Structured programming based on program sub-units (*functions*, *subroutines* and *modules*) enables
  - testing and debugging separately
  - re-use of code
  - improved readability
  - re-occurring tasks
- The key to success is in well defined data structures and scoping, which lead to clean procedure interfaces

What are procedures?

- With procedures we mean *subroutines* and *functions*
- Subroutines exchange data through its argument lists

```
call mySubroutine(arg1, arg2, arg3)
```

- Functions return a value

```
value = myFunction(arg1, arg2)
```

- Both can also interact with the rest of the program through module (global) variables

Declaration

Function

```
[TYPE] FUNCTION func(arg1,
arg2,) [RESULT(arg3)]
   [declarations]
   [statements]
END FUNCTION func
```

- Call convention

```
res = func(arg1, arg2, ...)
```

Subroutine

```
SUBROUTINE sub(arg1, arg2,...)
   [declarations]
   [statements]
END SUBROUTINE sub
```

- Call convention

```
CALL sub(arg1, arg2,...)
```

Declaration

```
real function dist(x,y)
 implicit none
 real :: x, y
 dist = sqrt(x**2 + y**2)
end function dist
```

```
subroutine dist(x,y,d)
 implicit none
 real :: x, y, d
 d=sqrt(x**2+y**2)
end subroutine dist
```

```
program do_something
...
 r=dist(x,y)
...
```

```
program do_something
 ...
 call dist(x,y,r)
 ...
```

Procedure arguments

- Call by reference: Means that only the memory addresses of the arguments are passed to the called procedure
  - any change to argument changes the actual argument
- Compiler can check the argument types only if the interface is explict, i.e. compiler has information about the called procedure at compile time.
  - INTENT keyword adds readability and possibility for more compile-time error catching

INTENT keyword

- Declares how formal argument is intended for transferring a value
  - in: the value of the argument cannot be changed
  - out: the value of the argument must be provided
  - inout (default)
- Compiler uses INTENT for error checking and optimization

```
subroutine foo(x,y,z)
  implicit none
  real, intent(in):: x
  real, intent(inout) :: y
  real, intent(out)   :: z

  x=10  ! compilation error
  y=10  ! correct
  z=y*x ! correct
end subroutine foo
```

Summary

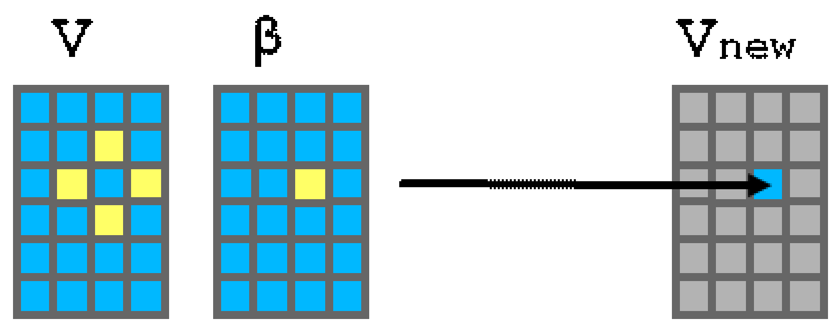
- Fortran is – despite its long history - a modern programming language especially for scientific computing
  - Versatile, easy to learn, powerful
- In our first encounter, we discussed
  - Variables & data types
  - Control structures: loops & conditionals
  - Operators
  - Program structuring with functions and subroutines

Programming assignment I

The Jacobi iterative scheme is a way of solving the 2D Poisson equation $\nabla^2 V=\beta$ by iteratively update the value of a 2D array V as

$$V_{new}(i,j)=[V(i-1,j)+V(i+1,j)+V(i,j-1)+V(i,j+1)-\beta(i,j)]/4$$

Until convergence has been reached (i.e. V_{new} and V_{old} are sufficiently close to each other).



Write a Fortran program that conducts the Jacobi iterative scheme. Return the program by email together with sample output by the next lecture.