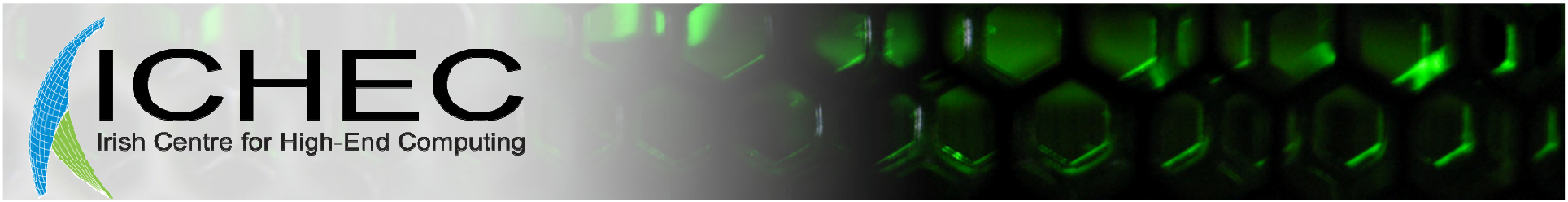


# Introduction to Modern Fortran

Alin M Elena (alin.elena@ichec.ie)
Computational Scientist


# Topics

1. General Programming Concepts
2. Source file rules
3. Intrinsic and user data types
4. Flow constructs
5. Functions and Subroutines
6. Modules
7. Arrays and array constructs
8. Dynamic allocation of memory

# Topics

9. Operator and subprograms overloading
10. I/O: formats, file and standard devices
11. Pointers
12. Preprocessing
13. Compile/Link/Debug
14. Command line arguments

# General Programming Concepts

## What is a program?

A program is the implementation of an algorithm in a specific programming language. (eg C, Fortran, C++, python)

## Generic elements of a program

1. Comments
2. Statements
3. Variables
4. Flow control
5. Subprograms
6. Modules

1. **Comments**: intended for the humans and totally ignored by the machine
2. **Statements**

- Control the flow of the program
- May resemble mathematical formulae (x=y+1)
- Executed one after another in the order in which they are written
- They are terminated by a special character
- Statements can be multiline

## 3. **Variables**

- a named sequence of memory locations to which a value can be assigned
- every variable has an address in the memory
- each variable must have a type (real, integer, character ...)
- variables should have meaningful names
- variables should be assigned a value before usage
- assignment operator "=": assigns the value of the expression from the right to the variable from the left
  x=2*sin(y)
- "=" is not restricted only to mathematical expressions
- A variable may not change its value during the program execution. We should call it a constant then.

## 4. Flow control

**Loops** are constructs that repeat a certain sequence of statements.

They can be with finite or unknown number of iterations.

```
do i=begin,end,increment
  <statements>
end
```

<statements> can be as complex as one wants and they may include other do constructs (nesting).

One should be able to abandon the loop or the current iteration if needed.

# Conditionals

A lot of times one may have to decide to execute a set of instructions for one situation and another for other situation. This is called branching.

```
If (logical
expression) then
   <statements 1>
else
  <statements 2>
endif
```

```
If (logical expression) then
   <statements 1>
endif
```

```
If (logical expression1) then
   <statements 1>
elseif (logical expression2) then
  <statements 2>
else
    <statements 3>
endif
```

<statements> can be as complex as one wants and they may include other if constructs.
<logical expression> should be any valid Boolean expression that evaluates to true or false.

## Loops II

### a priori conditioned loops

```
do while (logical expression)
  <statements>
enddo
```

The loop is executed as long the logical expression holds true.

The <statements> may not be executed at all if the logical expression does evaluate to false first time.

### a posteriori conditioned loops

```
do
  <statements>
  if (logical expression) exit loop
enddo
```

<statements> get executed at least once. The loop is executed as long as the logical expression evaluates to false.

For both constructs the number of iterations is unknown.

# Unconditional jumps

go to *label*

go to should be avoided and used only if no alternative exists. In day to day programming goto can be avoided almost every time.

## 5 Subprograms

- well defined parts of a programming language. They contain sequences of statements that can be called again and again in a program
- they can be functions or subroutines
- a subprogram takes a set of arguments and can return a result, change the values of arguments or simply do some I/O.

```
real function distance(x,y)
  distance = sqrt(x*x+y*y)
end function
```

```
subroutine distance(r,x,y)
      r = sqrt(x*x+y*y)
   end subroutine
```

# 6 Modules

Program entities that package subprograms, variables, constants so that they can be easily reused.

In addition almost any programming language should offer: input/output facilities and memory management features.

# Source file rules

1. Digits, letters, underscore and special characters form the fortran character set
2. A lower case letter is equivalent with the upper case counterpart so:
   call MyFunction  is the same as
   CALL MYFUNCTION
3. the underscore is a valid character in a name
4. maximum length of a name is 63 characters
5. There are two source forms: **fixed** and **free**. They should not be mixed in the same programming unit

Free source

- No restrictions on where a statement may appear on a line
- Lines may have zero characters
- A line may have at most 132 characters
- Blanks can appear with a meaning only in a character context or a format specifier
- Tokens can be separated by as many blanks you want. In these situations more blanks are considered one blank
- At least one blank should be used to separate names, labels, constants from other entities

- Some keywords may miss the blank: end do, end if, end function... (enddo, endif, endfunction)
- ! marks the start of a comment
- & at the end of the line but before any ! marks continuation on the next line
- ; and <enter> marks a statement termination
- no statement may start with a digit
- only 255 continuation lines are allowed for one statement

Fixed source*

- lines are limited to 72 characters
- the same rules on blanks as free form
- ! marks a comment except when is in a string or in column 6
- C or * in column 1 mark a comment
- Column 6 marks if a line is a continuation or not; blank or zero mean new statement
- Columns 7-72 may contain statements
- Col 1-5 may contain only labels if they are not comments

* avoid using it…

- include <source file>

an include line would make the compiler to physically insert the content of

<source file> in that place

- The source files are identified by the compiler by their extension.

  .F77, .f77, .F, .f, .for, .FOR – fixed form
  .f90, .F90, .f95, .F95, .f03, .F03 – free form

```
alin@blue:...sourcefiles>cat hello.f90
program hello
implicit none
! this is a simple program

   print *, "Hello World !!!   !"  ! just a print

end program hello
```

```
alin@blue:...sourcefiles>cat hello.f
      program hello
C this is a comment line
!23456
      implicit none

      print *, "hello world !!!  !" ! just a comment
      end
```

# Intrinsic and user data types

- Data types are general categories that describe the data that a program can use
- Data types can be intrinsic or user defined
- Data types can be used to declare constants, variables, pointers, targets (last two not so common)

**Logical** has only two values .true. and .false.

```
logical :: a
a=.true.
```

**character** keeps one character from the character set of the operating system. It can be any character.

```
character :: a
character(len=100) :: b
a="A"
b="This is a string!!!"
```

## Numerical data types

In fortran there are 3 intrinsic numerical types:

- integer
- real
- complex

All of them have an optional argument kind that allows the user to specify the precision.

```
integer (kind=4) :: a
real(kind=8) :: b
complex(kind=8) :: c
```

please note that the use of kind obsoletes the **double precision** and **double complex** types from old fortran

to select an integer in the range -10^9, 10^9

```
long=selected_int_kind(9)
integer (kind=long) :: a
 a=10_long
```

for a real with 15 digits precision and exponent range +/- 307 (equivalent of the double precision)

```
 double=selected_real_kind(15,307)
 real(kind=double) :: b
  b=1.0_double
```

In a lot of situations the intrinsic types are not convenient to describe your data. Let us think about storing data about a person. For each person you need to know name, age, weight. A nice and convenient way is to use user defined types. In Fortran would be

```
Type :: personType
  Character(len=100) :: name
  logical :: isMarried
  integer :: age
  real :: weight
end type personType
```

```
type(personType) :: a
a%name=”John Smith”
a%isMarried=.false.
a%age=25
a%weight=75.00
```

```
alin@blue:...datatypes>cat types.f90
module myTypes
implicit none
  private
  integer, parameter, public :: &
kpr=selected_real_kind(15,307)

  type, public :: personType
    character(len=100) :: name
    logical :: isMarried=.false.
    integer :: age
    real(kpr) :: weight
  end type personType

end module myTypes
```

```
alin@blue:...datatypes>cat test.f90
program test
  use myTypes
  implicit none
  real(kpr) :: a
  complex(kpr) :: c
  type(personType) :: d

  a=10.0_kpr
  c=cmplx(1.0_kpr,2.0_kpr,kpr)
  d%name="John Smith”
  d%isMarried=.true.
  d%age=25; d%weight=75.5_kpr


end program test
```

# Flow constructs

## comments

! marks the beginning of a comment <end of line> the end of the comment

Good practice mean that you should document your code. Add comments explaining what your program, piece of code does rather than how it does it.

## Expressions & statements

Variable = expression

Variable can be of any intrinsic or derived type

("=" has to make sense for the derived type)

Expressions are usually mathematical expressions. They are combinations of operators (arithmetic or relational) and operands.

An expression is evaluated according to the precedence rules of operators

**Operators**

Addition +
Subtraction -
Multiplication *
Division /
Exponentiation **
Expression grouping ()
Concatenation //

+,- can be binary or unary operators
a-b, a+b -> binary
-a, +a -> unary

**Relational operators**

equal to ==
not equal /=
less than <
less or equal <=
greater >
logical not .NOT.
logical and .AND.
Logical inclusive or .OR.
logical exclusive or .XOR.
Logical equivalent .EQV.
logical not equivalent .NEQV.

## Precedence of operators

(), **, * /, + -(unary), + - (binary), //, == /= < <= > >=, .NOT., .AND. .OR. .EQV. .NEQV.

Parenthesis can be used to change the precedence of operators

```
a=1
b=2
c=a+b**3     ! c=9
d=(a+b)**3   ! d=27
```

```
alin@blue:...expressions>cat types.f90
module myTypes
implicit none
  private
  integer, parameter, public :: &
      kpr=selected_real_kind(15,307)
end module myTypes
```

```
alin@blue:...expressions>cat test.f90
program test
  use myTypes
  implicit none
  real(kpr) :: a,b,c,f
  integer :: i,j,k
  character(len=100) :: name,lastName,d

  a=10.0_kpr; b=15.0_kpr
  c=sqrt(a*a+b**2)
  i=10; j=4
  k=i/j
  f=real(i,kpr)/real(j,kpr)
  name="John"; lastName="Smith"
  d = trim(name)//" "//trim(lastName)
  print *,c,k,d,f

end program test
```

```
alin@blue:...expressions>gfortran -o
exe types.f90 test.f90
alin@blue:...expressions>./exe
   18.027756377319946                2
John Smith
2.5000000000000000
```

# Flow control statements

| | |
|---|---|
| Conditional execution | **If/endif** |
| Conditional alternates | **Else/elseif** |
| Loop known no of iterations | **Do/enddo** |
| Loop unknown no of iterations | **Do while/enddo** |
| Terminate and exit loop | **exit** |
| Skip the rest of current iteration | **Cycle** |
| conditional case selection | **Select case/end select** |
| Stop execution | **Stop** |
| I/O statements | **Read/print/write** |
| Conditional array action | **Where/elsewhere** |
| Logical or array | **Any** |

# Conditionals

```
If (test) then
  <statements 1>
else
  <statements 2>
end if
```

```
If (test) then
  <statements>
end if
```

```
If (test1) then
    <statements 1>
elseif (test2) then
    <statements 2>
else
    <statements 3>
end if
```

```
If (test) <statement>
```

This program finds the real roots of a*x**2+b*x+c=0

```
alin@blue:...flow>cat test.f90
program test
  use myTypes
  implicit none
  real(kpr) :: a,b,c,d

  a=1.0_kpr; b=-3.0_kpr; c=2.0_kpr
  d= b*b-4*a*c
  if (abs(d)<tiny(1.0_kpr)) then
   print *, -b/(2.0_kpr*a)
  else
   print *, (-b+sqrt(d))/(2.0_kpr*a),(-b-sqrt(d))/(2.0_kpr*a)
  endif
end program test
```

# Loops

```
do index=b,e,i
   <statements>
end do
```

```
do while (test)
   <statements>
end do
```

```
do
   <statements>
   if (test) exit
end do
```

```
program loop1
 implicit none
 integer :: i,n=10
 do i=1,n
   print *,i,i**2
 end do
end program loop1
```

```
program loop2
 implicit none
 integer :: i,n=10
 i=1
 do while (i<=n)
   print *,i,i**2
   i=i+1
 end do
end program loop2
```

```
program loop3
 implicit none
 integer :: i,n=10
 i=1
 do
   print *,i,i**2
   i=i+1
   if (i>n) exit
 end do
end program loop3
```

**Implied loops** – a convenient way of writing loops
very compact
**(object, index=b,e,i)**
they can appear only in

- Read actions
- Print, write actions
- Data variable definition
- Definition of array elements

They cannot be nested

```
print *, (i*i,i=1,n,1)
```

would print the squares of all integers from 1 to n

## Nested loops and exit

```
do i=1,n
 do j=1,m
  <statements>
  if (test) exit
 end do
end do
```

Here exit will exit the closest enclosing loop (j for us). To exit the i loop one should use the labels

```
main: do i=1,n
 sec:  do j=1,m
  <statements>
  if (test) exit main
 end do sec
end do main
```

**cycle**

```
do i=1,n
   <statements 1>
   if (test)  cycle
   <statements 2>
end do
```

when condition test is true <statements 2> is skipped and the next iteration is started.

```
program loopExit
implicit none

 integer :: i,j,n=10

 do i=1,n
  do j=1,n
   if (i*j> 50 ) exit
   print *,i,j
  enddo
 enddo
end program loopExit
```

```
program loopExit
implicit none

 integer :: i,j,n=10

main: do i=1,n
sec:  do j=1,n
    if (i*j> 50 ) exit main
    print *,i,j
   enddo sec
 enddo main
end program loopExit
```

```
program loopCycle
  implicit none
  integer ::i, n=10

  do i=1,n
   print *,"processing: ",i
   if (mod(i,2)==0) then
    print *,"skip this one"
    cycle
   endif
   print *,i,i*i
  enddo
end program loopCycle
```

## Select

Select is a switching construct allowing the user to write long sequences of if/elseif/endif in a nice and compact way. Cases are selected according to an integer or character expression

```
select case(expression)
 case (value1)
   <statements 1>
 case (value2)
    <statements2>
  ...
 case (valueN)
   <statements N>
 case default
   <statements defaul
end select
```

The same as for do constructs the select construct can be named and used in the closing end.

```
program selectEx
  implicit none
  integer :: i

  print *,"input integer: "
  read(*,*)i
  select case (mod(i,2))
   case (0)
    print *,"even integer"
   case (1)
    print *,"odd integer"
  end select

end program selectEx
```

# Functions and Subroutines

```
subroutine s(a1,a2,...,an)
   type, intent :: a1,a2,..., an
  <statements>
end subroutine s


function f (a1,a2,..., an)
 type :: f
 type, intent :: a1,a2,..., an
  <statements>
 f=<expression>
end function f
```

```
program function1
  use myTypes
  implicit none

  real(kpr) :: a,b
  a=1.0_kpr; b=2.0_kpr;
  print *,dist(a,b)

contains
 function dist(x,y)
  real(kpr) :: dist
  real(kpr), intent(in) :: x,y

  dist=sqrt(x*x+y*y)
end function dist
end program function1
```

```
program function2
  use myTypes
  implicit none

  real(kpr) :: a,b
  a=1.0_kpr; b=2.0_kpr;
  print *,dist(a,b)

contains
 function dist(x,y) result(d)
  real(kpr) :: d
  real(kpr), intent(in) :: x,y

  d=sqrt(x*x+y*y)
 end function dist
end program function2
```

```
program function3
  use myTypes
  implicit none

  real(kpr) :: a
  a=1.0_kpr;
  print *,dist(a,2.0_kpr)

contains
real(kpr) function dist(x,y)
real(kpr), intent(in) :: x,y

  dist=sqrt(x*x+y*y)
end function dist

end program function3
```

```
program sub1
  use myTypes
  implicit none

  real(kpr) :: a,b,r

  a=1.0_kpr; b=2.0_kpr
  call dist(r,a,b)
  print *,r

contains
 subroutine dist(r,x,y)
  real(kpr), intent(in) :: x,y
  real(kpr), intent(out) :: r

  r=sqrt(x*x+y*y)
 end subroutine dist
end program sub1
```

```
program sub2
  use myTypes; implicit none
  real(kpr) :: a,b,r,a1,b1
  a=1.0_kpr; b=2.0_kpr
  a1=1.0_kpr; b1=0.0_kpr
  call dist(r,a1,b1,a,b);   print *,r
  call dist(r,x=a,y=b);   print *,r
contains
 subroutine dist(r,x1,y1,x,y)
  real(kpr), intent(in) :: x,y
  real(kpr), optional, intent(in) :: x1,y1
  real(kpr), intent(out) :: r
  if (present(x1)) then
    r=sqrt((x-x1)*(x-x1)+(y-y1)*(y-y1))
  else
    r=sqrt(x*x+y*y)
  endif
 end subroutine dist
end program sub2
```

# Modules

- Are program units that can encapsulate variables, constants, and/or subprograms for easy reuse in the main program or other program units.
- Good practice says that you should have one module per file.

```
module algebra
 use myTypes
 implicit none
 private
 integer, parameter, public :: kN=100
 integer :: M=200
 integer, public, save :: a
 public :: dist
 contains
 subroutine dist(r,x,y)
   real(kpr), intent(inout) :: r
   real(kpr), intent(in) :: x,y
   r=sqrt(x*x+y*y)
 end subroutine dist
end module algebra
```

```
program testModule
 use myTypes, only : kpr
 use algebra
 implicit none
 real(kpr) :: x1,y1,r
 x1=1.0_kpr;y1=2.0_kpr;
 call dist(r,x1,y1); print *,r
 a=30
end program testModule
```

# Arrays and array constructs

arrays are **indexed sets of values** of the same type

```
integer :: a(10) ! one dimensional array of integers with 10 values
integer :: b(10,10) ! two dimensional array 10 rows and 10 columns
```

7 is the maximum number of dimensions for an array in Fortran

By default the first element is a(1), b(1,1)

```
integer :: a(-1:8)
```

in memory the arrays are kept in contiguous linear chunks.

a 2d array is stored column by column from the first to the last row

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \textit{is } (1,4,2,5,3,6)$$

```
a=10 ! will set all elements of a to 10
b=9 ! will set all elements of b to 9
a(3:5)=11 ! will set a(3), a(4), a(5) to 11
b(1,:)=12 ! will set all elements from row 1 to 12
```

an array can be reshaped

```
integer :: a(2,3)
a=reshape((/1,2,3,4,5,6/),(/2,3/)) or
a=reshape((/1,2,3,4,5,6/),shape(a))
```

$$a = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

```
integer :: a(2,3), b(10,10)
a=b(1:2,1:3)
a=b(4:5,6:8)
```

# One can use full B:E:I construct to play with the arrays

$$A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

$$B = A(:, 2:1:-1) = \begin{pmatrix} 3 & 1 \\ 4 & 2 \end{pmatrix}$$

$$B = A(2:1:-1, :) = \begin{pmatrix} 2 & 4 \\ 1 & 3 \end{pmatrix}$$

Fortran has a rich set of intrinsic functions that act on arrays matching mathematical functions

a=b+c;

if c is a scalar would be added to all elements of a

if c a vector element by element addition. This holds for -,*,/,**, too.

c=transpose(a)

c=conjg(a)

c=matmul(a,b)

c=dot_product(a,b)

**array constructs:** all, any, count, maxloc, maxval, merge, minloc, minval, pack, unpack, sum, product, where

```
where (test)
   <assignments>
elsewhere
   <assignments>
end where
```

```
integer :: x(5)
x=(/-1,2,-5,-10,3/)
where (x<0)
  x=-x
end where
```

```
integer :: x(5)
x=(/-1,2,-5,-10,3/)
do i=1,5
  if(x(i)<0) then
    x(i)=-x(i)
  endif
enddo
```

```
integer :: x(5)
x=(/-1,2,-5,-10,3/)
print *,any(x<0) ! returns true
print *, all(x>0) ! returns false
```

the arrays constructs do not represent new concepts. They are just convenience statements for user.

# Passing arrays to subprograms

```
! assumed shape
 subroutine pass1(x)
  integer, intent(inout) :: x(:)
  x=10
 end subroutine pass1
```

```
!Adjustable size
 subroutine pass3(x,n)
  integer, intent(inout) :: x(n)
  integer, intent(in) :: n
  integer :: i
  do i=1,n
   x(i)=10
  enddo
 end subroutine pass3
```

```
! assumed size
 subroutine pass2(x,n)
  integer, intent(inout) :: x(*)
  integer, intent(in) :: n
  integer :: i
  do i=1,n
   x(i)=10
  enddo
 end subroutine pass2
```

# Dynamic allocation of memory

- We do not know the size of the array before running the program
- We want to minimize the usage of memory
- We want to reuse memory

allocate/deallocate/allocated

Static arrays by default go to stack

Dynamic arrays go to heap

good practice says that the number of allocate statements should be equal with the number of deallocate statements

```
integer, allocatable :: a(:),
b(:,:)
integer :: info, n=15,m=10
allocate(a(n),b(n,m),stat=info)
a=10
b=35
deallocate(a,stat=info)
allocate(a(n/2),stat=info)
a=b(1:7,5)
deallocate(a,b,stat=info)
```

**one can check if an array is allocated**

```
integer, allocatable, dimension(:) :: a
if (.not. allocated(a)) then
  allocate(a(n))
  a=0
else
  deallocate(a)
  allocate(a(n))
  a=0
endif
```

## automatic arrays

```
integer function test(n)
   integer, intent(in) :: n
        real :: a(n)
....
end function test
```

a is an automatic array and it exists only during the lifetime of the function test.

As automatic array get created and destroyed at entry/exit they may be a very expensive business.

# Operator and subprograms overloading

Modules offer another useful option: Overloading.

- Using the same name for different subprograms which have different numbers of arguments of different types
- Extending the usage of some operators

```
interface AddMatrix
  module procedure AddMatrixv1, AddMatrixv2
end interface
```

AddMatrixv1 and AddMatrixv2 are normal subprograms enclosed in the same module.

```
interface operator (+)
  module procedure myFunction
end interface
```

all arythmetic and relational operators can be overloaded. You can create your own using .XXX., where XXX is your operator.

```
interface assigment (=)
  module procedure MySubroutine
end interface
```

interface is used to create explicit interfaces for functions which are not included in a programming unit.

```
module matrix
use myTypes
implicit none
type, public :: matrixType
   integer :: n,m
   real(kpr), allocatable :: a(:,:)
end type matrixType
interface assignment (=)
  module procedure equalMatrixType
end interface
interface operator (+)
  module procedure addMatrixType
end interface
```

```
contains

 function addMatrixType(a,b) result(c)
  type(matrixType), intent(in) :: a,b
  type(matrixType) :: c
  allocate(c%a(a%n,a%m))
  c%a=a%a+b%a;   c%m=a%m;   c%n=a%n
 end function addMatrixType

 subroutine equalMatrixType(left, right)
  type(matrixType), intent(in) :: right
  type(matrixType), intent(inout) :: left
  left%a=right%a;   left%m=right%m;   left%n=right%n
 end subroutine equalMatrixType
end module matrix
```

# I/O: formats, file and standard devices

Read/write/print

```
read(*,*) <variable>
write(*,*) <variable/constant>
print *, is equivalent to write(*,*)
in read(*,*) first * means standard input
in write(*,*) first * means standard output
```

in both second * means no format assumed

format can be specified in place or at a later time using labels

```
write(*,'(2(i0,1x))')i,j
```

or

```
write(*,101)i,j
101 Format(2(i0,1x)))
```

```
Integer :: i=10
write(*,'(a10,i0)')"i0 ",i
write(*,'(a10,i5)')"i5 ",i
write(*,'(a10,i5.3)')"i5.3 ",i
write(*,'(a10,i1)')"i1 ", i
      i0 10
      i5    10
    i5.3   010
      i1 *
```

```
real :: b=10.043
complex :: c=cmplx(1.02, 3.04)
write(*,'(a10,f16.8)') "f16.8 ",b
write(*,'(a10,e16.8)') "e16.8 ",b
write(*,'(a10,e16.8 E3)') "e16.8 E3",b
write(*,'(a10,d16.8)') "d16.8 ",b
write(*,'(a10,en16.8)') "en16.8 ",b
write(*,'(a10,en16.8 e3)') "en16.8 e3",b
write(*,'(a10,es16.8)') "es16.8 ",b
write(*,'(a10,es16.8 e3)') "es16.8 e3",b
write(*,'(a12,2(F16.8,1x))')"2(F16.8,1x)", c
```

```
    f16.8       10.04300022
   e16.8  0.10043000E+02
e16.8 E3 0.10043000E+002
   d16.8  0.10043000D+02
  en16.8 10.04300022E+00
en16.8 e310.04300022E+000
    es16.8  1.00430002E+01
 es16.8 e3 1.00430002E+001
2(F16.8,1x)      1.01999998      3.03999996
```

```
logical :: l=.true.
write(*,'(a10,l4)')"l4", l
write(*,'(a10,g16.8 e3)')"g16.8 e3", i
write(*,'(a10,g16.8 e3)')"g16.8 e3", b
write(*,'(a10,g16.8 e3)')"g16.8 e3", l
write(*,'(a10,g16.8 e3)')"g16.8 e3", "characters"
        l4   T
 g16.8 e3                  10
 g16.8 e3  10.043000
 g16.8 e3                   T
 g16.8 e3     characters
```

```
program internal
implicit none
character(len=100) :: s1,s2
integer :: a,info
real :: b
s1="  10   40.5"
read(s1,*,iostat=info) a,b
a=a+1; b=b-1.0
write(s2,'(a2,i0,1x,a2,f16.8)')"a=",a,"b=",b
write(*,'(a50)',advance="no")trim(s2)
write(*,*)"this goes on the prev line”
end program internal
```

```
a=11 b=     39.50000000 this goes on the prev line
```

## more on units

Units are integers for example * in read is expanded to 5, in write to 6

One can redirect output to standard error using unit 0.

write(777,*) i

If unit 777 is associated with a file, i is printed in it, if not file fort.777 is created and i printed there

# I/O from files

Check if a file is in use - inquire

Associate an unit with a file – open

Release the unit when is not needed anymore – close

In a file one can write ascii or binary.

```
program file
implicit none
logical :: isopen,isit
integer :: myunit
inquire(file="myfile.dat",exist=isit,opened=isopen,number=myunit)
print *, isit, isopen, myunit
open(101,file="myfile.dat",status="unknown", action="write")
inquire(file="myfile.dat",exist=isit,opened=isopen,number=myunit)
print *, isit, isopen, myunit
write(101,*)"my first line in a file"
close(101)
end program file
```

```
 T F         -1
 T T         101
alin@blue:...io>cat myfile.dat
 my first line in a file
```

Status = old,new,scratch, replace, unknown

Action = read,write, readwrite

Position = asis, rewind, append

Form= formatted, unformatted

When used in unformatted form read/write statements should not have a format specifier.

```
program filebin
implicit none

integer :: a=10,b,info
open(101,file="myfile.dat",status="unknown", action="write",
     form="unformatted")
write(101)a
close(101)
open(101,file="myfile.dat",status="old", action="read",
     form="unformatted")
read(101,iostat=info)b
close(101)
write(*,*)b,b+1

end program filebin
```

```
program fileasci
implicit none
integer :: a=10,b,info

open(101,file="myfile.dat",status="unknown", action="write")
write(101,*)a
close(101)
open(101,file="myfile.dat",status="old", action="read")
read(101,*,iostat=info)b
close(101)
write(*,*)b,b+1
end program fileasci
```

Binary files

- cannot be understood directly by humans
- Keep all the precision for numbers
- Are machine dependent (big_endian, little_endian)
- Are smaller than the ascii counterparts

# Pointers

**Pointer**: the address of a data item

- Pointers contain no data, they just point to where the data is stored
- Fortran pointers do not have the same meaning as the c pointers. They are synonyms or aliases to variables
- Used to access portions of arrays or pass data through reference
- Suitable for dynamic data structures (lists, queues, stacks)
- The data to which they point is called a target
- A pointer has a logical status that marks its association or association with a target

```
program pointers
  implicit none
  integer, pointer :: a,b
  integer, target :: i,j,k
  i=100; j=50; k=300
  a=>i ! a points to i
  b=>j
  a=a+b ! i=i+j
  print *,i,j ! 150,50
  a=>k ; a=a+b ! k=k+j
  a=>null()
  print *, associated(a) ! false
  print *,associated(b,i) ! false
  print *, associated(b,j) ! true
end program pointers
```

# Preprocessing

preprocessing is a powerfull tool that can help you to turn portions of code on and off according to compile time parameters.

A file that contains preprocessing statements will have its extension in capitals

.F,.F77, .F90, .F95, F03 in fortran. That would determine the Fortran compiler to preprocess the file.

A fortran file can be also preprocessed by an external tool (eg cpp, fpp)

We will use preprocessing to enable/disable a second line of printing in the code.

```
program test
implicit none
  write(*,*)"Hello World!!!"
#ifdef MORE
  write(*,*)"From Alin"
#endif
end program test
```

```
alin@blue:...preprocessing> gfortran -DMORE -o hellof.x hello.F03
alin@blue:...preprocessing> ./hellof.x
 Hello World!!!
 From Alin
alin@blue:...preprocessing> gfortran -o hellof.x hello.F03
alin@blue:...preprocessing> ./hellof.x
 Hello World!!!
```

# Macro expansion

```
program macro
implicit none
#define func(x) (x*x+x)
#define one (x*x)
real :: y,x
  y=10.0; x=1.0
  print *, func(y)*10+y+one
end program macro
```

Nice and convenient for writing code but it may be a pain to read

# Compile/Link/Debug

## Compile

Transforms a source file (text file) into an object file(.o).

the compiler preproceses the file according to the rules with specified, if any.

validates the source file.

Generates the object file

```
ifort –c –o myobject.o test.f90
```

if no –o option is specified test.o is generated

```
alin@blue:...compileLinkDebug> ifort  -c test.f90
alin@blue:...compileLinkDebug> ls
test.f90 test.o
alin@blue:...compileLinkDebug> nm test.o
0000000000000000 r LITPACK_0.0.1
0000000000000000 T MAIN__
0000000000000008 r STRLITPACK_0.0.1
0000000000000010 r STRLITPACK_1.0.1
0000000000000020 r _2il0floatpacket.5
                 U __intel_new_proc_init
                 U for_set_reentrancy
                 U for_write_seq_lis
0000000000000000 b test_$A
```

**Linking** transforms an object file .o into an executable file by adding to it system libraries and user libraries.

ifort -o test test.o

alin@blue:...compileLinkDebug>ls

test test.f90 test.o

alin@blue:...compileLinkDebug> ldd test

linux-vdso.so.1 => (0x00007fff9a5ff000)
libm.so.6 => /lib64/libm.so.6 (0x00007f904ce42000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f904cc26000)
libc.so.6 => /lib64/libc.so.6 (0x00007f904c8cb000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f904c6b4000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f904c4b0000)
/lib64/ld-linux-x86-64.so.2 (0x00007f904d097000)

one can try to do a nm on the executable. all the symbols should be listed... pretty unreadable, isn't it?

**Debug:** the process of fixing the things

```
alin@blue:...compileLinkDebug> ifort -o test
  -debug all test.f90
alin@blue:...compileLinkDebug> ./test
         100         100         100         100         100
           0
alin@blue:...compileLinkDebug> ifort -o test -check all
      -traceback test.f90
test.f90(8): error #5561: Subscript #1 of the array A has
      value 6 which is greater than the upper bound of 5
  print *,a(6)*c
----------^
compilation aborted for test.f90 (code 1)
```

```
program test
implicit none
  integer :: a(5),c
  a=10
  call square(a)
  print *,a
  print *,a(6)*c
contains
  subroutine square(b)
    integer :: b(:)
    a=a*b
  end subroutine square
end program test
```

```
program test
implicit none
  integer :: a(5),c
  a=10
  call square(a)
  print *,a
  print *,a(5)*c
contains
  subroutine square(b)
    integer :: b(:)
    a=a*b
  end subroutine square
end program test
```

```
alin@blue:...compileLinkDebug> ./test
       100       100       100       100       100
forrtl: severe (193): Run-Time Check Failure. The variable 'test_$C' is being
      used without being defined
Image              PC                Routine            Line        Source
test               0000000000404710AD  Unknown          Unknown  Unknown
test               000000000046FBB5  Unknown            Unknown  Unknown
test               0000000000420600  Unknown            Unknown  Unknown
test               000000000403E9F  Unknown             Unknown  Unknown
test               0000000000404048B8  Unknown          Unknown  Unknown
test               0000000000402D09  MAIN__                 8  test.f90
test               0000000000402C2C  Unknown            Unknown  Unknown
libc.so.6          00007F6C04486B7D  Unknown            Unknown  Unknown
test               0000000000402B29  Unknown            Unknown  Unknown
```

# Command line arguments

A very common occurrence in C/C++ they do not appear in fortran standard up to 2003 version. You may still find compilers that do not support them or use some ad-hoc extensions.

Your friends:

GET_COMMAND,

COMMAND_ARGUMENT_COUNT

GET_COMMAND_ARGUMENT

```
program commandLine
implicit none
integer :: count,i
character(len=255) :: cmd
character(len=25) :: argum
call get_command(cmd)
write (*,*) trim(cmd)
count = command_argument_count()
write(*,*)"No of arguments: ", count
do i =1, count
  call get_command_argument(i, argum)
  write(*,'(a,i0,a)') "argument no ",i, " is: "//trim(argum)
enddo
end program  commandLine
```

```
alin@blue:...commandLine>./test  2003 577
    889 inp
 ./test 2003 577 889 inp
 No of arguments:           4
argument no 1 is: 2003
argument no 2 is: 577
argument no 3 is: 889
argument no 4 is: inp
```