Science and Technology Facilities Council

Scientific Computing

Introduction to Modern Fortran

Alin M Elena

ukri stfc daresbury laboratory

CoSeC
Computational Science Centre for Research Communities

CCP5
Collaborative Computational Project
Computer Simulation of Condensed Phases

Data-driven Materials and Molecular Science

July 15-16, 2024

Introduction

General Programming Concepts

Source file rules

Intrinsic and user data types

Flow constructs

Functions and subroutines

Modules

Arrays and array constructs

Dynamical allocation of memory

Overloading

Input/Output

Pointers

Preprocessing

Compile/Link/Debug

Command line arguments

Random numbers

Extras

Why?

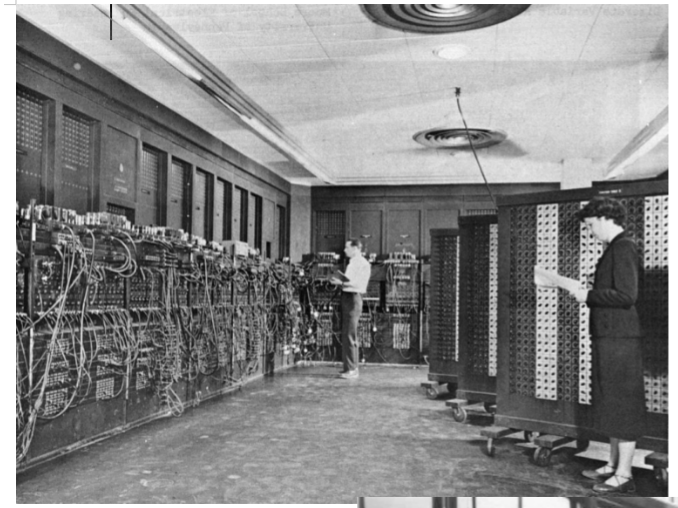
- In the good old days physicists repeated each other's experiments, just to be sure. Today they stick to FORTRAN, so that they can share each other's programs, bugs included.

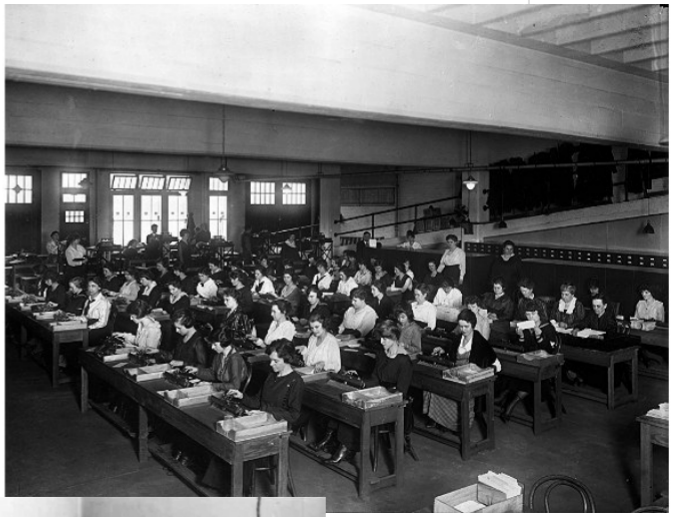
What

- Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer.
- Simplicity is prerequisite for reliability.
- The easiest machine applications are the technical/scientific computations.

How do we tell truths that might hurt? E W Dijkstra, 18 June 1975
https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF

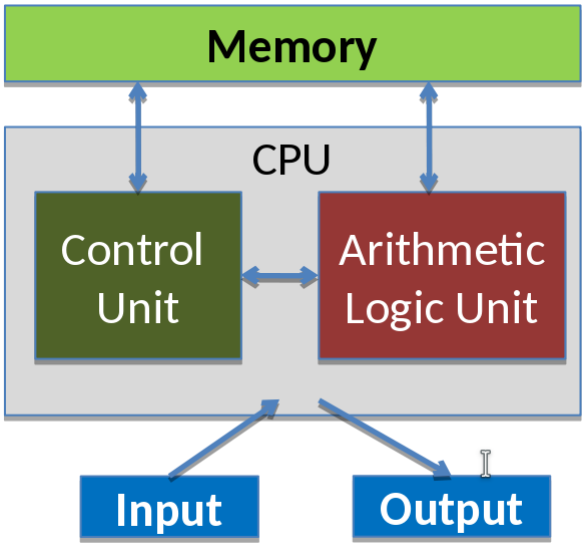
History





ENIAC, Computers, Mecanno

von Neumann Architecture



Four main components

- Memory
- Control unit
- Arithmetic Logic Unit
- Input/Output

First draft of a report on EDVAC, 1945, http://goo.gl/DnWs3t

History of Fortran

- FORmula TRANslation - FORTRAN, these days Fortran
- Fortran I-IV, 1953-1961, public released 1957 with IBM 704 by John Backus
- Fortran 66, 1966 first standard, X3.9-1966
- Fortran 77, !X3J3/90.4, ISO 1539:1980
- Fortran 90, ISO/IEC 1539:1991
- Fortran 95, ISO/IEC 1539-1:1997
- Fortran 2003, ISO/IEC 1539-1:2004(E)
- Fortran 2008, ISO/IEC 1539-1:2010
- Fortran 2018, ISO/IEC 1539:2018
- Fortran 2023, under vote to be published in October

https://gcc.gnu.org/wiki/GFortranStandards
https://wg5-fortran.org/N2201-N2250/N2212.pdf
https://j3-fortran.org/doc/year/23/23-007r1.pdf

Features

FORTRAN 77 added:

- DO loops with a decreasing control variable (index).
- Block if statements IF ... THEN ... ELSE ... ENDIF. Before F77 there were only IF GOTO statements.
- Pre-test of DO loops. Before F77 DO loops were always executed at least once, so you had to add an IF GOTO before the loop if you wanted the expected behaviour.
- CHARACTER data type. Before F77 characters were always stored inside INTEGER variables.
- Apostrophe delimited character string constants.
- Main program termination without a STOP statement.

Fortran 90 added

- Free format source code form (column independent)
- Modern control structures (CASE & DO WHILE)
- Records/structures - called “Derived Data Types”
- Powerful array notation (array sections, array operators, etc.)
- Dynamic memory allocation
- Operator overloading
- Keyword argument passing
- The INTENT (in, out, inout) procedure argument attribute
- Control of numeric precision and range
- Modules - packages containing variables and code

Fortran 95 added

- The FORALL statement and construct
- PURE user-defined procedures
- ELEMENTAL user-defined procedures
- Pointer initialization
- Derived-type structure default initialization
- Automatic deallocation of allocatable arrays
- CPU_TIME intrinsic subroutine
- Printing of -0.0
- Enhanced WHERE construct
- Zero-length formats
- user derived data-types, can have allocatable components

Fortran 2003 added

- Data enhancements and object orientation
  - Parameterized derived types
  - Procedure pointers
  - Finalization
  - Procedures bound by name to a type
  - Type extension
  - Overriding a type-bound procedure
  - Enumerations
  - The VOLATILE attribute
- Input/output enhancements
  - Asynchronous input/output
  - Intrinsic function for newline character
  - Stream access input/output
- Interoperability with C

full changes https://wg5-fortran.org/N1601-N1650/N1648.pdf

Fortran 2008

- Coarrays
- submodules
- do concurrent
- Contiguous attribute
- Implied-shape array
- Long integers
- Omitting an allocatable component in a structure constructor
- Finding a unit when opening a file
- g0 edit descriptor
- Bessel, error and gamma functions, Euclidean vector norms

full changes https://wg5-fortran.org/N1801-N1850/N1828.pdf

Fortran 2018

- Further interoperability of Fortran with C
- Additional parallel features in Fortran
- Conformance with ISO/IEC/IEEE 60559:2011

obsolescences

- common and equivalence
- Labelled do statements
- Specific names for standard intrinsic functions
- The forall construct and statement

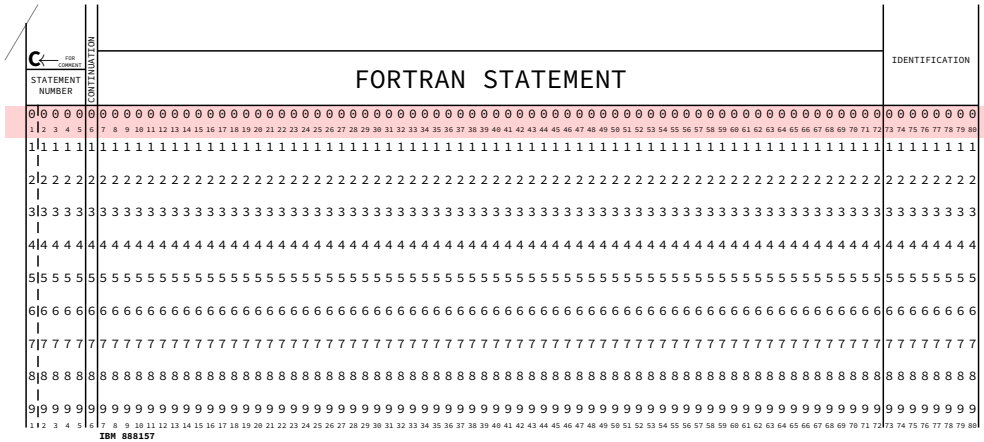
full changes

https://isotc.iso.org/livelink/livelink?func=ll&objId=19441669&objAction=Open&viewType=1

Fortran 2023

- lines longer than 132
- improvements to co-arrays
- enumerations
- integer arrays to specify subscripts and section subscripts
- no new obsoleted features
- https://wg5-fortran.org/N2201-N2250/N2212.pdf

Hello world!




```
1 program hello
2   implicit none
3   print *, ”hello world”
4 end program hello
```

References

- Standard Fortran 2018 http://isotc.iso.org/livelink/livelink?func=ll&objId=19442438&objAction=Open
- Michael Metcalf, John Reid, Malcolm Cohen, Modern Fortran Explained, 5th Edition, Oxford University Press
- Markus Arjen,Modern Fortran in Practice, Cambridge University Press
- Norman S. Clerman, Walter Spector, Modern Fortran, Cambridge University Press
- Walter S. Brainerd, Guide to Fortran 2008 Programming, Springer; 2nd ed. 2015
- Milan Curcic, Modern Fortran Building efficient parallel applications, Manning, 2020
- https://fortran-lang.org/
- gcc collection of standards https://gcc.gnu.org/wiki/GFortranStandards

Practicals for afternoon http://ccp5.gitlab.io/summerschool/

Programs

What is a program?

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

Generic elements of a program

- Comments
- Statements
- Variables
- Flow control
- Subprograms
- Modules

Comments&Statements

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

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.

Variables - data types

- an attribute of a variable
- an abstract description of how data is represented in memory
- predefined types or
- user defined types
- dictate what kind of operators/expressions/statements can be employed on variables.
- describe integers, real numbers, complex, characters, etc...

Flow control - loops

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.

More loops

A priori conditioned loops:

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

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.

More loops...

A posteriori conditioned loops:

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

<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.

Flow control - 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
```

<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.

Conditionals

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

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

Unconditional jumps

```
goto label1
     <some statements>
label1: point we jump to.
```

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

E Dijkstra, Go To Statements Considered Harmful, Communications of the ACM, 11(3),1968 https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf

Andrew Koenig http://www.drdobbs.com/cpp/what-dijkstra-said-was-harmful-about-got/228700940

...

“For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** stattements in the programs they produce.”

E Dijkstra, 1968

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
```

Modules

- Program entities that package subprograms, variables, constants so that they can be easily reused.
- Typically break your program into 1 module per file.
- can have submodules
- In addition almost any programming language should offer: input/output facilities and memory management features.

Fortran characters

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

Free source (modern)

- modern but not new... around since Fortran90
- 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 (ancient)

- 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...

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
```

literals

- type to represent literal data

```
character(len=42) :: a, b="12",&
       c="john's", d='h'
```

```
└~}X~}12                    john's                    h
```

intrinsic functions comparing strings

- LGE(A,B)
- LGT(A,B)
- LLE(A,B)
- LLT(A,B)

logicals

- boolean type which has only two values *.true.* or *.false.*

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

- default kind for a *logical* is the same as for *integer*

via iso_fortran_env

predefined kind values

- int8, int16, int32, int64,
- real32, real64, real128

all available kinds can be found using

- real_kinds, logical_kinds,character_kinds,integer_kinds
- atomic_int_kind, atomic_logical_kind

character_kinds can be used in kind for characters, but they have poor support in compilers for anything beyond ASCII (kind=1)

User defined types

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
```

Type bound procedures

```
type :: personType
  Character(len=100) :: name
  logical :: isMarried
  integer :: age;real :: weight
  contains
  procedure :: init
  final :: cleanup
end type personType

call a%init()
```

first parameter needs to be of **class(personType)** for procedures

first parameter needs to be of **type(personType)** for final

Implicit none

- In the dim and ancient past, FORTRAN did not need types to be specified.
- You could just use the variables, and FORTRAN assumed a type.
- **You Still Can. Just Don't.**

```
a = 1.2
i = 2
r = 2.0
```

- The rules were (and are):
- Anything starting: I,j,k,l,m,n are **implicitly assumed integer**.
- Other variables **assumed to be real**. No sizes specified.

Implicit

- You can change this behaviour using “implicit”

```
implicit integer (i-n), real (c-k), type(mine) (a,b)
```

- This is bad practice and leads to errors. Instead, do

```
implicit none
```

- before variable declarations to turn off implicit behaviour.
- Any undefined variables after that will cause an error.

Variable attributes: volatile

Variable definitions can have attributes: one example here is volatile

```
integer, volatile :: a
```

Rarely used in Fortran code (seen in C/C++ more often) it means a variable may be changed outside the scope of the Fortran program; i.e. by another program, hardware (its a memory register), etc.

Used to tell the compiler to avoid optimisations.

Variable attributes: save

- The save attribute specifies that a local variable of a program unit or subprogram retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement unless it is a pointer and its target becomes undefined
- If it is a local variable of a subprogram it is shared by all instances of the subprogram.

```
integer, save :: a
integer :: b=10
```

Variable attributes: parameter

- marks a variable as constant

```
integer, parameter :: a = 10
```

Variables scope

the scope of a variable restricts to the programming unit is declared in it: module, program, function or subroutine

```
k = 10
block

  integer :: k

  k=1
end block
!k is still 10
```

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.

Hint: Old FORTRAN could not have blank lines. So you will see lines beginning with 'C' for comment but containing nothing.

replace them with a blank line for clarity.

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.

Operators

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
```

Bit wise

- **bge**(i, j) Bitwise greater than or equal to.
- **bgt**(i, j) Bitwise greater than.
- **ble**(i, j) Bitwise less than or equal to.
- **blt**(i, j) Bitwise less than.
- i,j integer or boz literal constant
- ishft(i,shift)
- ishftc
- ior, ieor and iand
- not
- iall, iany, iparity

Examples

```
! save me in types.F90
module myTypes
 use iso_fortran_env, only : real64
 implicit none
 private
 integer, parameter, public :: &
  kpr=real64
end module myTypes
```

...

```
! save in 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
```

...

```
$ gfortran -o exe types.f90 test.f90
$ ./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, All*

Conditionals

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

if (test) then
 <statements>
end if

if (test) <statement>
```

...

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

Example

```
! save in roots.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)<epsilon(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
```

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 (valueN)
   <statements N>
  case default
   <statements defaul
end select
```

The select construct can be named and used in the closing end.

Example

```
program selectEx
  implicit none
  integer :: i
  Integer, parameter :: EVEN=0, ODD=1

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

end program selectEx
```

Loops

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

```
program loop1
  implicit none
  integer :: i,n=10

  do i=1,n
    print *,i,i**2
  end do
end program loop1
```

Loops 2

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

```
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
```

Loops 3

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

```
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

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

....

```
print *, (i*i,i=1,n,2)
```

would print the squares of all odd integers from 1 to n

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

Nested loops & 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

Nested loops & exit...

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

lexit=.false.
do i=1,n
  do j=1,m
    <statements>
    if (test) then
      lexit=.true.
      exit
    end if
  end do
  if (lexit) exit
end do
```

Examples

```
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
    end do
  end do
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
```

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.

Example

```
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
```

Introduction

```
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
```

function styles 1

```
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
```

function styles 2

```
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
```

function styles 3

```
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
```

subroutines 1

```
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
```

subroutines 2

```
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
```

recursive

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0 \quad F_1 = 1$$

```
recursive integer function fibonacci(n) result(r)
  integer, intent(in) :: n

  if (n<2) then
    r=n
  else
    r=fibonacci(n-1)+fibonacci(n-2)
  endif
end function fibonacci
```

pure

- Pure functions/subroutines, depend only the data provided by the list of arguments
- for functions all arguments need intent(in)
- for procedures arguments can be intent(in|out)
- side effects only for arguments which contain intent out
- may give freedom to compiler to optimize
- one cannot do IO
- one cannot call procedures which are not pure
- are mandatory if to be issued in a **do .. concurrent**

```
pure function dist(x1,y1,x,y)
pure subroutine dist(r,x1,y1,x,y)
```

elemental

- scalar operators
- scalar dummy argument
- scalar return value
- but they can be invoked on arrays
- in the array case are applied element wise
- Elemental subroutines, side effects are permitted via arguments.
- elemental procedures are usually *pure* if not the case use *impure* to mark it as such

```
elemental real function test3(x)
  real, intent(in) :: x
  test3 = x*x
end function test3

real :: z(4)=[1.0,2.0,3.0,4.0]
print *, test3(z)
```

Intrinsic

- Fortran contains a large variety of intrinsic functions and subroutines
- check the standard (Fortran 2008), chapter 13, page 315 for a complete set

...

- 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.
- can have submodules, not discussed in here.

using modules 1

```
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
```

using modules 2

```
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 (I)

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)

can be of any intrinsic or user defined type

Arrays (II)

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 (column major)

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

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

Arrays (III)

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))
```

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

```
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

$$a = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

$$b = a(:, 2:1:-1) = \begin{matrix} 2 & 1 \\ 4 & 3 \end{matrix}$$

$$b = a(2:1:-1, :) = \begin{matrix} 3 & 4 \\ 1 & 2 \end{matrix}$$

...

```
integer :: a(2,2),b(2,2),i
a = reshape([1,3,2,4],[2,2])
do i=1,2
  write(*,'(i0,1x,i0)') a(i,:)
enddo

b=a(:,2:1:-1)
do i=1,2
  write(*,'(i0,1x,i0)') b(i,:)
enddo

b=a(2:1:-1,:)
do i=1,2
  write(*,'(i0,1x,i0)') b(i,:)
enddo
```

Specifying data and arrays

```
x = (/ 1.0, 2.0, 3.0, 4.0, 5.0 /) !or
x = [ 1.0, 2.0, 3.0, 4.0, 5.0 ]

a = 42.0_8*[(i/10.0_8,i=1,n)]
b = [(i/(1.0_8+i*i),i=1,n)]
c = [(sin(a(i)+2.0_8*b(i)),i=1,n)]
```

or you can use *data*. see the standard

Intrinsic Functions

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

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

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.

Dimension statement

This is the old form of how to describe arrays. The syntax

dimension a(10), b(20,40)

was used at the top of the program / subprogram (variable definitions).

Avoid this form. In modern Fortran, we use either:

```
real, dimension(10)    :: a
real, dimension(:)     :: b
real                   :: x(10)
```

Passing arrays assumed shape

```
subroutine pass1(x)
  integer, intent(inout), contiguous :: x(:)
  x=10
end subroutine pass1
```

When you pass an array with assumed shape, extent is lost

...

```
! assumed shape
real, dimension(-3:3) :: y,x
call pass1(x,y,start)

subroutine pass1(x,y,start)
  integer :: start
  real, intent(inout) :: x(-3:),y(start:)
end subroutine pass1
```

Passing arrays assumed size

```
! 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
```

Old style. In Fortran 77, Arrays were contiguous. They had size, but not shape.

Can't do whole array operations: x=10. The upper bound in the last dimension must appear in the reference to the assumed size array

Array needs to be contiguous. Reshaping I/O, are not safe.

Passing arrays adjustable size

```
! 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
```

size, bound functions

These give the size, upper bound, lower bound of a passed array.

- size(array,dim); lbound(array, dim); ubound(array,dim)

```
subroutine subtest(array)
  real(kind=8), allocatable, intent(in) :: array(:,:)
  integer                               :: iii, jjj

  if(allocated(array)) then
    print*, size(array, 1), size(array, 2)
    do iii = lbound(array, 1), ubound(array, 1)
      do jjj = lbound(array, 2), ubound(array, 2)
        print*, array(iii,jjj)
      enddo
    enddo
  endif
end subroutine
```

assumed rank arrays

assummed objects can be array or scalar

```
real :: a0
real :: a1(2)
real :: a2(3,3)
real :: a3(4,4)

call sub(a0); call sub(a1); call sub(a2); call sub(a3)

contains

  subroutine sub(a)
    real, intent(inout) :: a(..)
    print *, rank(a)
  end subroutine sub
```

for per rank operations use *select rank*

Introduction

- 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
- deallocate memory in reverse order of allocation, avoid memory fragmentation

Example

```
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.

Procedure 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.

Example

```
interface field_write
  module procedure field1d_write, field2d_write
end interface

!> Write a 1D field to its netCDF file
!@param iter Iteration
subroutine field1d_write(this, iter)
  implicit none
  type(field1d), intent(in) :: this
  integer, intent(in) :: iter

  call out_check( nf90_put_var( this%ncid, this%varid, &
     this%current))
end subroutine
```

...

```
!> Write out the current results to netcdf file
subroutine field2d_write(this, iter)
  integer, intent(in) :: iter
  type(field2d), intent(in) :: this

  integer :: start(3), count(3)

  !Write one step of data
  count = [this%x, this%y, 1 ]  start = [ 1, 1, 1 ] ;  start(3) = iter

  call out_check( nf90_put_var( this%ncid, this%varid, &
      this%current, start = start, count = count))
end subroutine
```

Operator Overloading

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

all arithmetic 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 also used to create explicit interfaces for functions which are not included in a programming unit.

Example: matrix maths

```
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
```

read/write/print

```
read(*,*) <variable>
write(*,*) <variable/constant>
print *, <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))
```

read

```
integer :: a
real :: f
character(len=100) :: h="12.5 10"
write(*,*)"read integer: "
read(INPUT_UNIT,*)a
write(*,*)"read real: "
read(*,*) f
print *,a,f
read(h,*)f,a
print *,a,f
```

standard units

Fortran has two output statements: *print* and *write*. The print:

```
print *, “Hello world”
```

Write takes an output unit, format statement and contents

```
write (OUTPUT_UNIT, ‘(A10)’) my_name
write (*,*) “Hello world”
write (ERROR_UNIT,*) “Hello world”
```

use iso_fortran_env to get OUTPUT_UNIT, ERROR_UNIT and INPUT_UNIT

formatting

```
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 *
```

... reals

```
real :: b=10.043
complex :: c=cmplx(1.02, 3.04)
write(*,'(a12,2(F16.8,1x))')”2(F16.8,1x)”, c
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
```

...

```
2(F16.8,1x)       1.01999998       3.03999996
      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 e3 10.04300022E+000
     es16.8   1.00430002E+01
  es16.8 e3 1.00430002E+001
```

generals

```
logical :: l=.true.
write(*,'(a10,g16.8 e3)')”g16.8 e3”, “characters”
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
```

```
g16.8 e3       characters
      l4    T
g16.8 e3                      10
g16.8 e3   10.043000
g16.8 e3                       T
```

internal units

```
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 formats

```
program test
  implicit none

  integer, allocatable :: a(:)
  real, allocatable :: b(:)
  integer :: i,n=13,m=15

  allocate(a(n),b(m))
  a=[(i*i/3,i=1,n)]
  b=[(i/3.0,i=1,m)]
  call print_me_nice(a,b,n,m)

  write(*,'(*(g0," "))') b
  deallocate(a,b)

contains
```

...

```
subroutine print_me_nice(a,b,n,m)
  integer, intent(in) :: a(:),n,m
  real, intent(in) :: b(:)

  character(len=50) :: fma,fmb
  integer :: i

  write(fma,"(a,i0,a)") "(",n,"(i0,1x))"
  write(fmb,'(a,i0,a)'),"(",m,"(g0,1x))"

  write(*,trim(fma))(a(i),i=1,n)
  write(*,trim(fmb))(b(i),i=1,m)

end subroutine print_me_nice
end program test
```

more on units

- Units are integers for example * in read is expanded to INPUT_UNIT (5), in write to OUTPUT_UNIT (6) One can redirect output to standard error using unit ERROR_UNIT (0).
- the standard name constants are loaded by using **use iso_fortran_env**

```
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

writing to a file

```
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
```

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.

reading from a file

```
program fileascii
  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 fileascii
```

```
10           11
```

...

```
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
```

```
10           11
```

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

newunit

```
integer :: myunit

myunit = 10
open(newunit=myunit,file='test')
```

- automatically selects a unit number that does not interfere with other units already connected
- you will get a negative number different of -1 or any of the INPUT_UNIT, ERROR_UNIT, OUTPUT_UNIT
- if you get en error the value of myunit is unchanged

Introduction

- 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
```

Introduction

- preprocessing is a powerful tool that can help you to turn portions of code on and off according to compile time parameters.
- preprocessing is not standard in Fortran, apart of **include** statement
- it is widely supported by all modern compilers
- 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)
- an inluded file via **include** does not get preprocessed.
- 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
```

```
gfortran -DMORE -o hellof.x hello.F90
./hellof.x
  Hello World!!!
  From Alin
```

```
gfortran -o hellof.x hello.F90
./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, try to avoid.

Compiling

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

```
gfortran -c -o myobject.o test.f90
```

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

...

```
$  nm comp.o
                 U _gfortran_set_args
                 U _gfortran_set_options
                 U _gfortran_st_write
                 U _gfortran_st_write_done
                 U _gfortran_transfer_array_write
                 U _gfortran_transfer_integer_write
0000000000000286 T main
000000000000009e t MAIN__
0000000000000010 r options.10.3823
0000000000000000 t square.3791
```

Include paths

Fortran looks in directories for .mod files and include files that are required by the compile line

If adding code from multiple directories, or third-party libraries, use “-I <directory>” to include these, e.g.

```
gfortran -c  -I<path to mods> module.f90
```

This directory can include both include files and module files.

NB: If you include a blank “-I” without specifying a directory, it means “ignore the current directory”.

Linking

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

```
gfortran -o comp.X comp.o
ldd comp.X
        linux-vdso.so.1 (0x00007ffc6cd30000)
        libgfortran.so.5 => /usr/lib64/libgfortran.so.5 (0x00007fc71bbe6
        libm.so.6 => /lib64/libm.so.6 (0x00007fc71b853000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fc71b63b000)
        libquadmath.so.0 => /usr/lib64/libquadmath.so.0 (0x00007fc71b3fb
        libc.so.6 => /lib64/libc.so.6 (0x00007fc71b03b000)
        libz.so.1 => /lib64/libz.so.1 (0x00007fc71ae24000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fc71c055000)
```

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

Libraries

- Libraries are precompiled collections of code (sets of .o files)
- Typically named: libfoo.a or libbar.so

The compiler links against these using the syntax:

```
gfortran -o ./myprog prog.f90 -lfoo -lbar
```

If the library is not in the current directory, you need to specify the directory with “-L”

```
gfortran -o ./myprog prog.f90 -L<path to foo> -lfoo \
-L<path to bar> -lbar
```

...

you can compile and link in one go

```
gfortran -o comp.X comp.F90
```

be careful at compilation errors versus linking errors.

- errors relating to syntax are compilation errors
- errors relating to missing functions are linking errors.

Debugging

Debug: the process of fixing the things

you can use special tools like gdb or the compiler via flags.

```
-g -Wextra -frecord-gcc-switches -O0 -std=f2008 -pedantic -fbacktrace
-fcheck=all -finit-integer=2147483647
-finit-real=snan -finit-logical=true -finit-character=42
-finit-derived -ffpe-trap=invalid,zero,overflow
-fdump-core -fstack-protector-all -Wall -pipe
```

other useful flags -fsanitize

Example

```
program test
implicit none
  integer :: a(5)
  real :: 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
```

...

```
gfortran -O0 -std=f2008 -pedantic -fbacktrace -fcheck=all \
-finit-integer=200000 -finit-real=snan -finit-logical=true \
-finit-character=42 -finit-derived -ffpe-trap=invalid,zero,overflow \
-g comp.F90 -o comp.X
./comp.X
       100           100           100           100           100
Program received signal SIGFPE: Floating-point exception -
  erroneous arithmetic operation.
Backtrace for this error:
#3  0x400bdc in test
        at ../comp.F90:12
#4  0x400c59 in main
        at ../comp.F90:12
Floating point exception (core dumped)
```

Introduction

A very common occurrence in C/C++/Python they do not appear in Fortran standard up to 2003 version.

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
```

...

```
./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
```

...

```
program random
  implicit none
  integer(kind=4), allocatable :: seed(:)
  real(kind=8) :: r(2)
  integer :: is,i,p

  is = 13
  call random_seed(size=p)
  allocate(seed(p))
  seed = 17*[(i-is,i=1,p)]
  call random_seed(put=seed)
  deallocate(seed)
  call random_number(r)

  print *,r
end program random
```

More

- interoperability with C
- co-arrays
- submodules
- oop features
- ieee floating point standard conformance

Software engineering

- use libraries, do not reinvent the wheel (unless you are a wheel maker)
- use a version control system, git is one of the most popular and *de facto* standard in open source world (git-scm.org).
- use a build system for your project. Quite few out there, *cmake* is one of the popular ones used by some scientific projects. (cmake.org), meson
- test your code, use one of the continuous integration platforms (gitlab-ci, travis, jenkins, ...) also check the coverage of your code
- use tools to instrument code
- have a coding style document and contribution guide.

gprof

```
gfortran -o hello.x -p hello.F90
./hello.x
```

a new file *gmon.out* is created

```
gprof ./hello.x
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 48.81      0.18     0.18        1   180.58   321.03  MAIN__
 37.96      0.32     0.14 200000000     0.00     0.00  __mymath_MOD_my
  6.78      0.35     0.03                             __do_global_dtor
  6.78      0.37     0.03                             frame_dummy
```

Some Bad patterns

go to

Did I mention not to use *go to* already?

- sometimes rewritting code with *go to* to structured form can be a pain in the neck.
- reimplementing the algortihm may be much simpler and faster.

one routine for many things

```
subroutine answer_to_all(...,stage)

   block 1 of code
   if stage == 1
      block 2 of code
   else
      block 3 of code
   end if
```

...

```
subroutine block_1
   block 1 of code

 subroutine block_2
   block 2 of code

 subroutie block_3
   block 3 of code
```

use named constants instead of cryptic maps

```
if vdw == 0 then
   block 1
elseif vdw == 1 then
   block 2
elseif vdw == 2 then
   block 3

energy = energies(1)+energies(3)
```

...

```
integer, parameter :: POT_LJ=0
integer, parameter :: POT_MORSE=1
integer, parameter :: POT_BUCK=2

integer, parameter :: ENER_KINETIC=1
integer, parameter :: ENER_POTENTIAL=3

if vdw == POT_LJ then
   block 1
elseif vdw == POT_MORSE then
   block 2
elseif vdw == POT_BUCK then
   block 3

energy = energies(ENER_KINETIC)+energies(ENER_POTENTIAL)
```

modules are not common blocks

- do not use modules as a replacement for common blocks.
- keeping variables in modules and sharing them via them looks like a great idea
- until you try to use more than one core for your programme.

Q&A

<..>

$$x_{n+1} = 108 - \frac{815 - \frac{1500}{x_{n-1}}}{x_n}$$

$$x_0 = 4$$

$$x_1 = \frac{17}{4}$$

$$x_{42} = ?$$