Introduction to Fortran95 and the Unix Environment

in Large-Scale and Supercomputing Course

June 1 - 5, 2015

Napsu Karmitsa

Department of Mathematics and Statistics
University of Turku, Finland

Contents

- **Fortran95**
  - Introduction & Motivation
  - Data Types & Variable Declarations
  - Operators
  - Free Source Form
  - Control Structures
  - Procedures & Modules
  - Arrays
  - Input/Output
- **The Unix Programming Environment**
  - Program compilation and linking: `make`
  - Program debugging: GNU debugger `gdb`
  - Program profiling: `gprof` (on Friday)

The Goal

- Attendees should be able to write small Fortran95 programs and understand legacy software written in Fortran.

Fortran95:

Introduction & Motivation

Fortran 95: Introduction and Motivation

- **Why Fortran?**
  - Well suited for numerical computations;
  - Fast code (in addition, compilers can optimize well);
  - Handy array data types;
  - Clarity of code;
  - Portability of code;
  - Large number of optimized numerical libraries build for Fortran;
  - A lot of legacy software written in it (usually FORTRAN77).
- **Why not Fortran?**
  - Personal taste;
  - Some things easier to implement e.g. in C or C++.

Introduction to F95: First Program

```
PROGRAM  test_example
  ! This is a comment. Comments start with an exclamation mark.
  ! Command PROGRAM starts the main program, END PROGRAM ends it.
  ! Some exponentiation and squareroot computations.

  IMPLICIT NONE ! Do not use implicit variable declarations.

  REAL :: x,y ! Data type declarations
  INTRINSIC SQRT ! f95 standard provides many commonly used functions.

  WRITE (*,*) 'Give a value for x:'  ! Ask a number and read it in.
  READ (*,*) x

  y = x**2 + 1 ! Power function and addition.

  WRITE (*,*) 'Given value for x: ', x
  WRITE (*,*) 'Computed value for x**2 + 1: ', y
  ! SQRT(y) returns the square root of the argument y.
  WRITE (*,*) 'Computed value for SQRT(x**2 + 1): ', SQRT(y)

END PROGRAM test_example
```

Fortran95:

Data Types & Variable Declarations

Data Types

- Fortran provides two kind of data types: *intrinsic data types* and *derived data types*. The five intrinsic data types are

  INTEGER, an integer number;

  REAL, a real number;

  COMPLEX, a complex number

  – a pair of real numbers used in complex arithmetic;

  LOGICAL, a logical value

  – TRUE or FALSE;

  CHARACTER, a string consisting one or more characters.

- Derived data type is a structure of data types which is defined by the programmer (more about this later).
- These data types are handled with Fortran operators, procedures and statements.

Variable Declarations

- **Variables** must be *declared* at the beginning of the program or procedure.
- After declaration the value of a *variable can be changed* whenever needed.

```
IMPLICIT NONE

INTEGER :: n0
INTEGER :: n1=2   ! Variables can also be initialized at their
                  ! declaration.

REAL :: a,b
REAL :: c=0.0, d=2.3E-6

COMPLEX :: imag_a
COMPLEX :: imag_b=(0.1, 1.0E+2)   ! imag_b = 0.1 + 100.0i

CHARACTER(LEN=10) :: cat1
CHARACTER(LEN=80) :: cat2='Emma the Cat'

LOGICAL :: test0= .TRUE.
LOGICAL :: test1= .FALSE.
```

Variable Declarations (Cont.)

- **Constants** are defined with PARAMETER clause.
- The value of the *constant can not be changed* after declaration.

```
REAL, PARAMETER :: pi=3.14159
CHARACTER(LEN=*), PARAMETER :: cat3='Rääpäle'
```

Array Declarations

- **Arrays** are declared in a pretty much similar fashion to scalar variables.

```
! 1 dimensional character array, not initialized at declaration
INTEGER, PARAMETER :: n_entries = 43, n = 7
CHARACTER(LEN=30), DIMENSION(n_entries) :: name

! 1 dimensional real arrays, not initialized
REAL, DIMENSION(n_entries) :: mark
REAL, DIMENSION(0:n-1) :: vector   ! By default, Fortran indexing starts at
                                   ! element number 1.

! 3 element 1 dimensional integer array, bounds defined, initialized
INTEGER, DIMENSION(-1:1) :: x = (/0, 1, 2/)

! Assigning values:
name(1)= 'George'
mark(1)= 10.0
name(2)= 'John'
mark(2)= 9.9
...
name(43)= 'Bill'
mark(43)= 4.1
```

- Later: more details about arrays.

KIND-Attribute for Data Types

- Variable precision can be declared using the KIND-statement.
- KIND-attribute is a compiler dependent unit.
- The corresponding values can be inquired by standard functions.

```
IMPLICIT NONE

! SELECTED_INT_KIND(r):  integer, range between -10**r < n < 10**r.
INTEGER, PARAMETER :: short=SELECTED_INT_KIND(4)

! SELECTED_REAL_KIND(p):  real, precision at least p decimals
! SELECTED_REAL_KIND(2*PRECISION(1.0)) is the old DOUBLE PRECISION.
INTEGER, PARAMETER :: prec=SELECTED_REAL_KIND(12)

! SELECTED_REAL_KIND(p,r):  real, range between -10**r < x < 10**r and
! precision at least p decimals
INTEGER, PARAMETER :: prec_plus=SELECTED_REAL_KIND(12,100)

INTEGER(KIND=short) :: i
REAL(KIND=prec) :: x,y
COMPLEX(KIND=prec) :: imag_a   ! can be used for complex variables as well

x=1.0_prec
y=2.0_prec * x**2
```

KIND-Attribute for Data Types (Cont.)

```
PROGRAM  pi

  IMPLICIT NONE

  INTEGER, PARAMETER :: sp=SELECTED_REAL_KIND(6,30), &
                        dp=SELECTED_REAL_KIND(12)

  REAL(KIND=db) :: pi1
  REAL(KIND=sb) :: pi2
  INTRINSIC ATAN

  pi1 = 4*ATAN(1.0_db)
  pi2 = 4*ATAN(1.0_sp)
  WRITE (*,*) 'pi:', pi1, pi2

END PROGRAM pi
```

- Output (depends on computer):

  `3.1415926535897 93 3.141593`

Numerical Precision for Data Types

- Some other intrinsic functions related to numerical precision:
  - `i=INT(a,kind)`, conversion to integer (rounding down)
  - `r=REAL(a,kind)`, conversion to real
  - `i=KIND(p)`, the kind of the supplied argument
  - `r=TINY(a)`, the smallest positive number
  - `*=HUGE(a)`, the largest positive number (INTEGER or REAL)
  - `r=EPSILON(a)`, the smallest number r such that $1 + r > 1$
  - `i=DIGITS(a)`, the number of significant digits
  - `i=MAXEXPONENT(a)`, the largest exponent
  - `i=MINEXPONENT(a)`, the smallest exponent
  - `i=RANGE(a)`, the range of exponent
  - `i=PRECISION(a)`, the decimal precision

Numerical Precision for Data Types (Cont.)

```
PROGRAM  test_precision
  ! You should get the precision you need (at least),
  ! your computer/compiler may give some extra.

  IMPLICIT NONE

  INTEGER, PARAMETER :: sp=SELECTED_REAL_KIND(6,30), &
                        dp=SELECTED_REAL_KIND(10,200)

  REAL(KIND=sp) :: a
  REAL(KIND=dp) :: b

  WRITE (*,*) sp, dp, KIND(1.0), KIND(1.0_db)
  WRITE (*,*) KIND(a), HUGE(a), TINY(a), RANGE(a), PRECISION(a)
  WRITE (*,*) KIND(b), HUGE(b), TINY(b), RANGE(b), PRECISION(b)

END PROGRAM test_precision
```

- Output:

```
4 8 4 8
4 3.4028235E+38 1.1754944E-38 37 6
8 1.79769313486231 6E+308 2.225073858507201E-308 307 15
```

FORTRAN95:

OPERATORS

Operators

Arithmetic operators

```
REAL :: x,y
INTEGER :: i = 10
x=2.0**(-i)          ! exponentiation and negation (1)
x=x*REAL(i)          ! multiplication and type change (2)
x=x/2.0              ! division (2)
i=i+1                ! addition (3)
i=i-1                ! subtraction (3)
                     ! a**2+b/d-3*-5 = ((a ** 2) + (b / d)) - (3 * (-5))
```

Relational operators

```
<   or .LT.  ! less than
<=  or .LE.  ! less than or equal to
==  or .EQ.  ! equal to
/=  or .NE.  ! not equal to
>   or .GT.  ! greater than
>=  or .GE.  ! greater than or equal to
```

Note! Do not compare real numbers with == operator. Rather use e.g. the statement

```
IF (ABS(x - a) < 10*EPSILON(a)*ABS(a)) THEN
```

Operators (Cont.)

Logical operators

```
.NOT.   ! logical negation (1)
.AND.   ! logical conjunction (2)
.OR.    ! logical inclusive disjunction (3)
.EQV.   ! logical equivalence (4)
.NEQV.  ! logical nonequivalence (4)
```

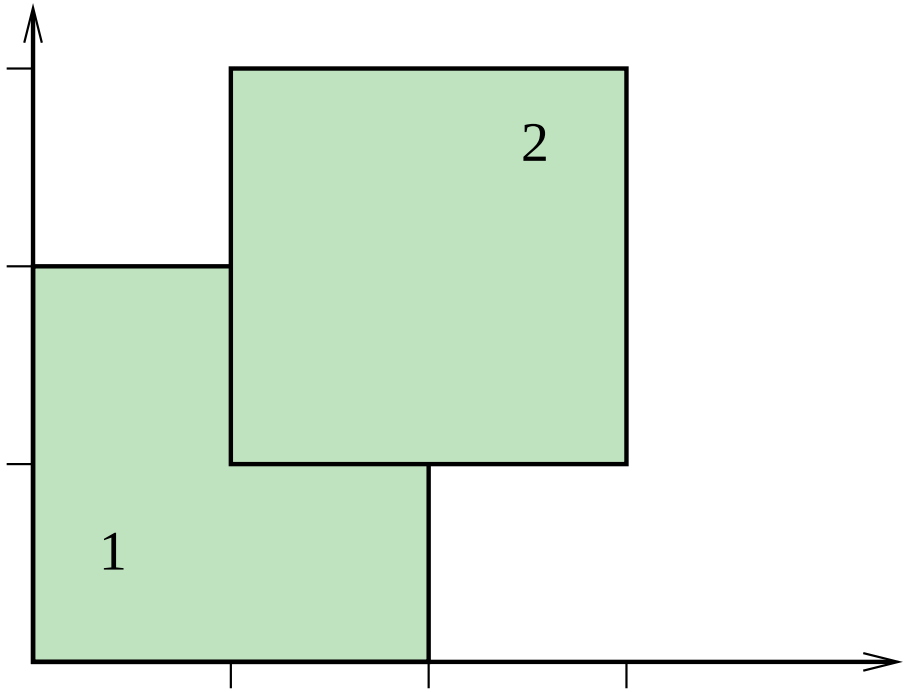
Logical operators `a` and `b`.

- `F` means `.FALSE.` and `T` means `.TRUE.`

a	b	a .AND. b	a .OR. b	a .EQV. b	a .NEQV. b
F	F	F	F	T	F
F	T	F	T	F	T
T	F	F	T	F	T
T	T	T	T	T	F

Operators Example

Placement of a point (x, y) in two overlapping area $[0, 2] \times [0, 2]$ and $[1, 3] \times [1, 3]$?



Operators Example

```
PROGRAM  placetest
  ! Test logical and relational operators.
  IMPLICIT NONE

  LOGICAL :: square1, square2
  REAL :: x,y

  WRITE (*,*) 'Give point coordinates for x and y:'
  READ (*,*) x, y

  square1 = (x >= 0. .AND. x <= 2. .AND. y >= 0. .AND. Y <= 2.)
  square2 = (x >= 1. .AND. x <= 3. .AND. y >= 1. .AND. Y <= 3.)

  ! This example has also the control structure IF ...
  IF (square1 .AND. square2) THEN   ! both are .TRUE.
    WRITE (*,*) 'Point within both squares'
  ELSE IF (square1) THEN   ! just square1 is .TRUE.
    WRITE (*,*) 'Point in square 1'
  ELSE IF (square2) THEN   ! just square2 is .TRUE.
    WRITE (*,*) 'Point in square 2'
  ELSE   ! both are .FALSE.
    WRITE (*,*) 'Point outside'
  END IF

END PROGRAM placetest
```

Numerical Standard Functions

- Fortran 95 -standard provides 115 commonly used functions or subroutines (intrinsic procedures), for example,
  - Numeric functions; `ABS, CEILING, FLOOR, INT, MAX, MIN, MOD, REAL, SIGN, ...`
  - Mathematical functions; `ACOS, ASIN, COS, EXP, LOG, LOG10, SIN, SQRT, ...`
  - Character functions; `ACHAR, ADJUSTL, ADJUSTR, LEN, LEN_TRIM, LGE, TRIM, VERIFY, ...`
  - Array manipulating functions; `ALL, ANY, DOT_PRODUCT, MATMUL, MAXVAL, RESHAPE, SUM, ...`
  - and many many more.
- It is a good practice to introduce the standard procedure used at the beginning of the program or procedure. This can be done by INTRINSIC -clause:

```
INTRINSIC ABS, SIN, SQRT
```

FORTRAN95:

FREE SOURCE FORM

F95: Free source form and other remarks

- **Variable name**
  - can not be longer than 31 characters,
  - only letters, digits or underscore are allowed,
  - must start with a letter,
  - no distinction between lower and uppercase characters.
- **Maximum row length** may be 132 characters.
- There may be 39 **continuation lines**:
  - if a line is ended with ampersand &, it will be continued on the next line.

```
! Continuation line example.
INTEGER :: a,b,c,d
a=a+b+ &
  c+d
! The above is equivalent to either of the following two lines.
a=a+b+c+d
A = A +b+c+D ! Fortran is not case sensitive.
```

F95: Free Source form and other remarks

- **Character strings** are case sensitive.

```
CHARACTER(LEN=32) :: string1,string2
LOGICAL :: ans
string1 = 'a'
string2 = 'A'
ans = string1 .EQ. string2
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.

! Ampersand & is a continuation mark only at the end of the line.
WRITE (*,*) 'In Fortran, & forces the compiler to treat two lines as&
            & one. This is accomplished by placing & at the end of&
            & the first line and at the beginning of the second line.'
```

F95: Free Source form and other remarks

- **Statement separation** can be done using newline or 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
```

- IMPLICIT NONE is one of the most important statement in F95
  - "God is real unless declared integer."
  - in older Fortrans (66,77) *implicit type definitions* were commonly used: variables beginning with `i,j,k,l,m,n` are integers and others are real,
  - if you leave out IMPLICIT NONE the same convention is used in Fortran95.

F95: Free Source form and other remarks

- **Automatic change of representation** (important Fortran feature)

```
PROGRAM  numbers
  IMPLICIT NONE

  COMPLEX :: c,cc
  REAL :: r
  INTEGER :: i

  i = 7.3                    ! same as i = INT(7.3)
  r = (1.618034, 0.618034)   ! same as r = REAL((1.618034, 0.618034))
  c = 2.7182818              ! same as c = CMPLX(2.7182818)
  cc = r*(1,1)               ! at first the variable r is changed CMPLX(r)
  WRITE (*,*) i, r, c, cc

END PROGRAM numbers
```

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

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

FORTRAN95:

CONTROL STRUCTURES

Control Structures

- **IF ... THEN ... ELSE** branching
- **SELECT CASE** selecting
- **DO** looping
- **GOTO** **Do not use!**

Control Structures: IF ... THEN ... ELSE

Conditional execution

```
[tag:]  IF (logical expression) THEN
   executable statements
[ELSE IF (logical expression) THEN [tag]
   executable statements]
[ELSE [tag]
   executable statements]
END IF [tag]
```

- In case there is only one statement after the IF construction, then the THEN ... END IF pair can be omitted: e.g.

```
IF (i > 10) a = 1.0
```

- In addition, the tags are optional.

```
PROGRAM  if_example
  IMPLICIT NONE

  REAL :: x,y,eps,t
  INTRINSIC ABS, EPSILON

  WRITE (*,*) 'Give x and y:'
  READ (*,*) x, y
  eps = EPSILON(x)

  IF (ABS(x) > eps) THEN
    t=y/x
  ELSE
    WRITE (*,*) 'division by zero'
    t=0.0
  END IF

  WRITE (*,*) 'y/x = ',t
END PROGRAM  if_example
```

Control Structures: IF ... THEN ... ELSE

```
PROGRAM  referenced_if_example
! It is possible to give tags for IF branches.
  IMPLICIT NONE

  REAL :: x,y,eps,t
  INTRINSIC ABS, EPSILON

  WRITE (*,*) 'Give x and y:'
  READ (*,*) x, y
  eps = EPSILON(x)

  outside: IF (ABS(x) > eps) THEN
    inside: IF (y > eps) THEN
      t=y/x
    ELSE IF (ABS(y) <= eps) THEN inside
      t=0.0
    ELSE inside
      t = -y/x
    END IF inside inside
    WRITE (*,*) 'result = ',t
  END IF outside

END PROGRAM  referenced_if_example
```

Control Structures: SELECT CASE

Conditional execution

```
 [tag:]  SELECT CASE (argument)
 [CASE (value[,value, ...])  [tag]
    executable statements]
 [CASE DEFAULT [tag]
    executable statements]
 END SELECT [tag]
```

- 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.
- If there is no CASE DEFAULT and no match found then the statement following END SELECT is executed.

```
...
INTEGER :: i
LOGICAL  :: isprimenumber

...
SELECT CASE (i)
  CASE (2,3,5,7)
    ! Variables are not allowed
    ! on the list.
    isprimenumber = .TRUE.
  CASE (1,4,6,8:10)
    ! case value range, form low:high.
    isprimenumber = .FALSE.
  CASE DEFAULT ! DEFAULT-branch.
    isprimenumber = testprimenumber(i)
      ! Function call.
END SELECT
...
```

Control Structures: DO -loops

Three different DO -loops

- **Plain DO -loop**

```
[tag:]  DO
    executable statements
END DO [tag]
```

- **Count controlled DO -loop**

```
[tag:]  DO index = initial, limit[, step]
    executable statements
END DO [tag]
```

- **DO WHILE -loop**

```
[tag:]  DO WHILE (test)
    executable statements
END DO [tag]
```

Control Structures: DO -loops

DO -loop with an integer counter (Count controlled)

```
PROGRAM  newt
  ! Newton iteration for finding the roots of
  ! function f(x) = e^x + x - 5.
  IMPLICIT NONE

  REAL :: x = 0.0
  INTEGER :: i, n = 20
  WRITE (*,*) 'iterations:  ', n
  WRITE (*,*) 'starting point x = ', x

  DO i = 1, n
    x = x - (EXP(x) + x - 5)/(EXP(x) + 1)
    WRITE (*,*) 'x = ', x
  END DO
END PROGRAM  newt
```

- In addition DO loop may have tags.
- You may select the initial and final value as well as the stepsize for DO-loop. E.g.

```
DO i = 20, 1, -1
```

Control Structures: DO -loops

DO -loop without a counter (Plain)

```
...
REAL :: x, totalsum
totalsum = 0.0
DO
  READ (*,*) x
  IF (x < 0.0) THEN
    EXIT ! exit the loop.
  ELSE IF (x == 0.0) THEN
    CYCLE ! cycle back to the beginning.
  END IF
  totalsum = totalsum + 1.0/SQRT(x)
END DO
...
```

- Inside loop controls: EXIT and CYCLE
- EXIT statement ends only a particular active loop (there can be nested loops = a loop within a loop)
- CYCLE tells not to do any more statements below this one in the active DO loop and cycle back to the beginning of the loop.

Control Structures: DO -loops

DO WHILE -construct (Condition controlled loop)

```
...
REAL :: x, totalsum
...
totalsum = 0.0
READ (*,*) x
DO WHILE (x > 0.0)
  totalsum = totalsum + x
  READ (*,*) x
END DO
...
```

Control Structures: DO -loops, Example

```
PROGRAM  newt
  ! Newton iteration for finding the roots of
  ! function f(x) = e^x + x - 5.  Second version.
  IMPLICIT NONE

  REAL :: x = 0.0, d
  REAL, PARAMETER :: toler = 1E3*EPSILON(x)
  INTEGER :: i, maxit = 20
  INTRINSIC ABS, EXP

  WRITE (*,*) 'starting point x = ', x

  DO i = 1, maxit
    d = (EXP(x) + x - 5)/(EXP(x) + 1)
    x = x - d
    WRITE (*,*) 'i = ', i ' x = ', x
    IF (ABS(d) <= toler*ABS(x)) EXIT
  END DO
END PROGRAM  newt
```

Control Structures: DO -loop and IF, Example

```
PROGRAM  gcd
  ! Computes the greatest common divisor, EUCLIDEAN ALGORITHM.
  IMPLICIT NONE

  INTEGER, PARAMETER :: long=SELECTED_INT_KIND(9)
  INTEGER(KIND=long) :: m, n, t
  INTRINSIC MOD

  WRITE (*,*) 'Give positive integers m and n:'
  READ (*,*) m, n
  WRITE (*,*) 'm:  ', m,' n:  ',n

  positivecheck: IF (m > 0 .AND. n > 0) THEN
    main_algorithm: DO WHILE (n /= 0)
      t = MOD(m,n) ! mod(n,m) gives the remainder when n is divided by m.
      m = n
      n = t
    END DO main_algorithm
    WRITE (*,*) ''Greatest common divisor:  ', m
  ELSE
    WRITE (*,*) 'Negative value entered.'
  END IF positivecheck
END PROGRAM gcd
```

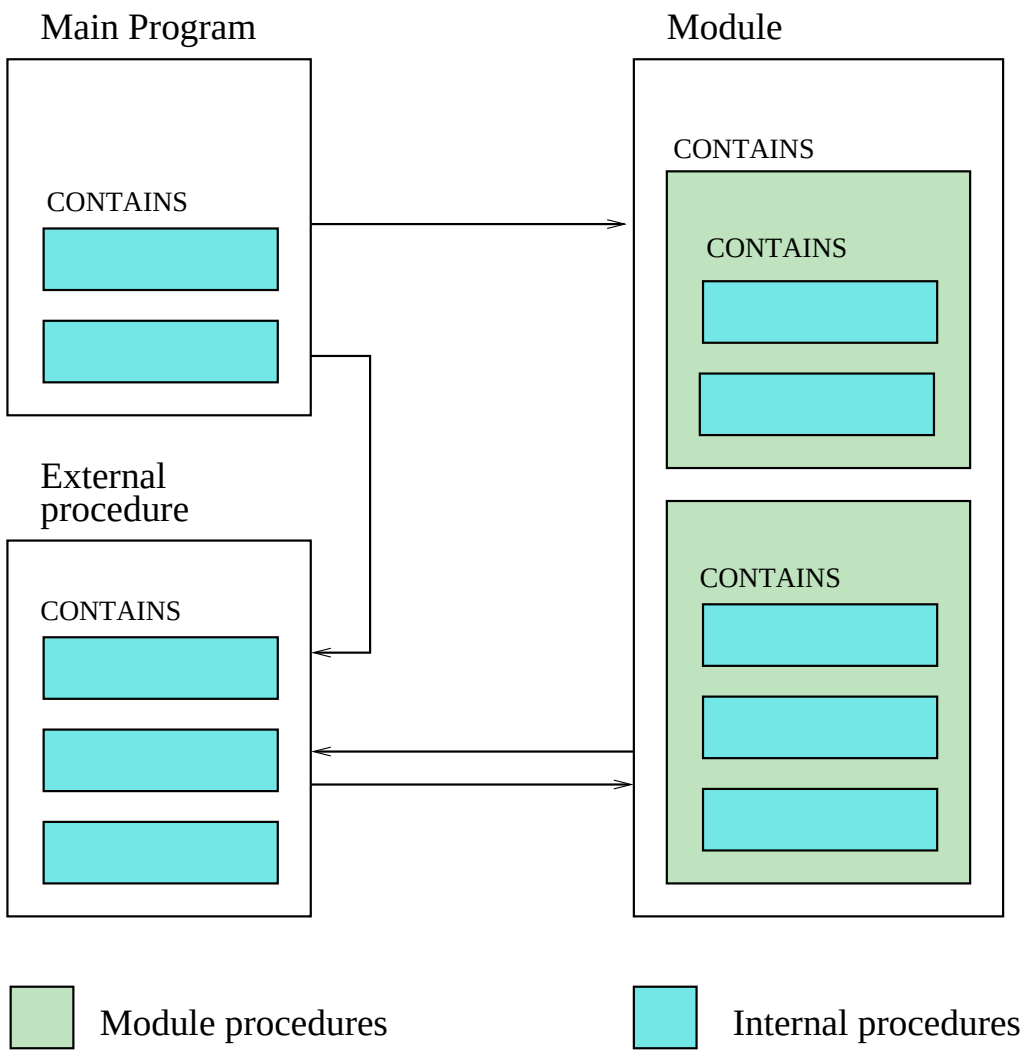
Fortran95:

Procedures & Modules

Procedures & Modules

- Procedures and modules allow structured programming. Advantages:
  - testing and debugging separately
  - recycling of code
  - improved readability
  - re-occurring tasks
- By procedures we mean subroutines and functions
- Many ready-to-use libraries with different procedures are available (BLAS, EISPAC, ISML, LINPACK, MINPACK, NAG, SPARSKIT ...).
- Modularity means dividing a program into small minimally dependent modules. Advantages:
  - Constants, variables, data types and procedures can be defined in modules
  - Makes possible to divide program into smaller selfcontained units

Program Units



Subroutines and Functions

- **Subroutines** exchange data with arguments only.
- **Functions** return value according to its declared data type.
- Declarations:

Function

```
[TYPE] FUNCTION func([arg1][,arg2,...])  [RESULT(arg)]
    [declarations]
    [executable statements]
END FUNCTION func
```

Function call:

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

Subroutine

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

Subroutine call:

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

Functions: Declaration Example

```
FUNCTION decimals(i) RESULT(decim)
  ! Number of decimals in given integer
  IMPLICIT NONE

  INTEGER :: i, decim
  INTEGER :: tmp
  INTRINSIC ABS

  decim = 1
  tmp = ABS(i)

  DO WHILE (tmp >= 10)
    tmp = tmp/10
    decim = decim + 1
  END DO

END FUNCTION  decimals
```

```
PROGRAM  deci
  IMPLICIT NONE

  INTEGER :: n
  WRITE (*,*) 'Give n: '
  READ (*,*) n

  WRITE (*,*) 'Number n is: ', n
  WRITE (*,*) 'Decimals: ', decimals(n)

  CONTAINS
    ! FUNCTION decimals is an internal
    !   procedure of PROGRAM deci

    Declaration of FUNCTION decimals

END PROGRAM  deci
```

Output:

```
Give n:
10101
Number n is: 10101
Decimals:  5
```

Subroutines: Declaration Example

```
SUBROUTINE transpose(a,b)
  ! change the places of two reals
  IMPLICIT NONE

  REAL :: a, b
  REAL :: tmp

  tmp = a
  a = b
  b = tmp

END SUBROUTINE transpose
```

```
PROGRAM  test_trans
  IMPLICIT NONE

  REAL :: x = 1.0, y = -1.0

  WRITE (*,*) 'x,y: ', x, y

  IF (x > y) THEN
    CALL transpose(x,y)
  END IF
  WRITE (*,*) 'x,y: ', x, y

  CONTAINS
    ! SUBROUTINE transpose is an internal
    !   procedure of PROGRAM test_trans

    Declaration of SUBROUTINE transpose

END PROGRAM  test_trans
```

Output:

```
x,y: 1.000000 -1.000000
x,y: -1.000000 1.000000
```

Procedure Arguments

- The lists of actual and formal arguments should match:
  - the number of arguments,
  - arguments types, and
  - the order in which the arguments are listed.
- Any change to a formal argument value changes the actual argument.
  - Do not alter values of function arguments!
- Compiler checks arguments if the interface of procedure is known at compilation time:
  - internal and module procedures.
- Use of arguments can be limited by **INTENT** -keyword.

INTENT -KEYWORD

- Declares how the formal argument is intended for transferring a value
  - INTENT(in)
  - INTENT(out)
  - INTENT(inout)
- Compiler uses this for error checking and optimization

```
SUBROUTINE func(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 = 10 ! Correct
  y = z ! Garbage/Compilation error
  z = y*x ! Correct

END SUBROUTINE func
```

Saving Local variables

- By default variables in procedures are dynamically allocated at invocation.
- Only saved variables keep their value from one call to the next: **SAVE** -attribute

```
PROGRAM squaresum
  IMPLICIT NONE

  WRITE (*,*) 'result: ', ssum(1.0)
  WRITE (*,*) 'result: ', ssum(2.0)

  CONTAINS
    REAL FUNCTION ssum(x)
    ! Note an alternative way of function declaration
      IMPLICIT NONE
      REAL, INTENT(in) :: x
      REAL, SAVE :: sum = 0.0
      sum = sum + x**2
      ssum = sum
    END FUNCTION ssum
END PROGRAM squaresum
```

Output:

```
result:  1.0000000
result:  5.0000000
```

Procedure Types

- Internal, external and module procedures
  - **Internal**: within program structure
  - **External**: independently declared, may be other language
  - **Module procedure**: defined in module
  - **Recursive procedure** that calls itself
- Internal and module procedures provide a defined interface, compiler uses this to check arguments

Internal Procedures

- Each program unit (host) may contain internal procedures.
  - Host may be the main program, external procedure or module procedure.
  - An internal procedure can not have internal procedures.
- Declared at the end of program unit after **CONTAINS** -statement.
- Inherits variables and objects from the host.

```
SUBROUTINE outer
  IMPLICIT NONE
  REAL :: x,y

  ...
  CONTAINS
    SUBROUTINE inner
      IMPLICIT NONE
      REAL :: y ! y is local variable
      y = x + 1 ! x is common with the host (outer) procedure
    END SUBROUTINE inner
END SUBROUTINE outer
```

External Procedures

- Relic from Fortran 77.
- Declared in separate program units.
- Modules are much easier and more secure.
- Library routines are external procedures.
- Program units written in different language (e.g. C, Matlab) are external procedures.
- **EXTERNAL** -keyword.

```
EXTERNAL outer ! for subroutine outer
REAL, EXTERNAL ::  outer2 ! for function outer2
```

Recursive Procedures

- Recursion means calling a procedure within itself
- **RECURSIVE** -keyword for compiler

```
RECURSIVE FUNCTION recurse(n) RESULT(facto)
! factorial n! of n
  IMPLICIT NONE

  INTEGER, INTENT(in) :: n
  INTEGER :: facto

  IF (n == 1) THEN
    facto = 1
  ELSE
    facto = n * recurse(n-1)
  END IF

END FUNCTION recurse
```

```
PROGRAM  factorial
  IMPLICIT NONE

  INTEGER :: n

  WRITE (*,*) 'Give n: '
  READ (*,*) n
  WRITE (*,*) 'n = ', n
  WRITE (*,*) 'n!  = ', recurse(n)

  CONTAINS

    Declaration of FUNCTION recurse

END PROGRAM  factorial
```

Output:

```
Anna n: 7
n = 7
n! = 5040
```

Modular Programming

- Modularity means dividing a program into small minimally dependent pieces (modules).
- Advantages:
  - constants, variables, data types and procedures can be defined in modules;
  - makes it possible to divide program into smaller self contained units.

Modules

- Global definitions
- The same procedures and data types available in different program units:
  - Procedures defined in modules can be used in any other program unit
  - Module procedures are declared after CONTAINS -statement
- Compile-time error checks:
  - Placing procedures in modules helps compiler to detect programming errors and to optimize the code
- Hide implementation details (OOP or object oriented programming)
- Define group routines and data structures
- Define generic procedures and custom operators

Module Structure

Declaration example:

```
MODULE accuracy
  IMPLICIT NONE

  INTEGER, PARAMETER :: &
    realp = SELECTED_REAL_KIND(6)
!     realp = SELECTED_REAL_KIND(12)
  INTEGER, PARAMETER :: &
    intp = SELECTED_INT_KIND(4)
END MODULE accuracy

MODULE check
  USE accuracy
  IMPLICIT NONE

  REAL(KIND=realp) :: y

  CONTAINS
  FUNCTION check_this(x) RESULT(z)
    INTEGER(KIND=realp) :: x, z
    z = HUGE(x)
  END FUNCTION check_this
END MODULE check
```

Declaration:

```
MODULE module_name
  [declarations]
  [CONTAINS
   module procedure
     [module procedures]...]
END MODULE module_name
```

Usage example:

```
PROGRAM testprog
  USE check
  IMPLICIT NONE

  REAL(KIND=realp) :: x,test

  test=check_this(x)

END PROGRAM testprog
```

It is also possible to limit variables with

```
USE accuracy, ONLY: realp
USE accuracy, ONLY: reaali => realp
```

Global Variables with Modules

- COMMON definitions in Fortran 77

```
INTEGER N,NTOT
COMMON/EQ/N,NTOT
REAL ABSTOL,RELTOL
COMMON/TOL/ABSTOL,RELTOL
```

- The same implemented with modules

```
MODULE globals
  INTEGER, SAVE :: n, ntot
  REAL, SAVE :: abstol, reltol
END MODULE globals
```

- With the SAVE -option variables are stored in memory throughout the program execution

Visibility of Objects

- Objects in modules can be PRIVATE or PUBLIC.
- Default is PUBLIC that is visible for all program units using the module.
- PRIVATE will hide the objects from other program units.

```
INTEGER, PRIVATE :: x
INTEGER, PUBLIC :: y ! PUBLIC is the default
```

or

```
PRIVATE
INTEGER :: x,y
PUBLIC ::  y
```

Derived Data Types and Modules

- Derived data type is a structure of data types which is defined by the programmer
- Comprises of any data types including other derived types
- Abstract data type includes data type definitions and procedures
- Type is declared in a declaration section of a program unit or a procedure
- Not visible to other programming units UNLESS defined in modules and used via USE -command.

Derived Data Types

Declaration:

```
TYPE [, privacy ::]  name_of_type
  [declarations]
END TYPE name_of_type
```

```
TYPE student_eval
  CHARACTER(LEN=30) :: name
  INTEGER :: st_number
  REAL :: mark
END TYPE student_eval
```

Usage:

```
TYPE(name_of_type)[dimension] :: &
  variable_name
```

```
INTEGER, PARAMETER :: nro = 50
TYPE(student_eval), DIMENSION(nro) :: &
  students
```

Elements are accessed using % -operator:

```
variable_name % component
```

```
students(1)%name = 'Marko'
students(1)%st_number = 78655
students(1)%mark = 9.8
! You can also use structure constructor
students(2) = student_eval('Aaro',65544,8.3)
```

FORTRAN95:

ARRAYS

ARRAYS

- Arrays enable a natural way to access vector and/or matrix data during computation
- Fortran language is a very versatile in handling especially multi-dimensional arrays (unlike C or some other languages)

Array Declaration and Syntax

- Arrays are declared in a pretty much similar fashion to scalar variables
- Arrays refer to a particular built-in data type (or derived data type), but they have **one or more dimensions** specified in the variable declaration
- Fortran95 supports **up to 7 dimensions**

```
INTEGER, PARAMETER :: m = 100, n = 500
INTEGER :: idx(m)
REAL(kind = 4) :: vector(0:n-1)
REAL(kind = 8) :: matrix(m, n)
CHARACTER(len = 80) :: screen (24)
TYPE(my_own_type) :: object (10)

! or

INTEGER, DIMENSION(1:m) :: idx
REAL(kind = 4), DIMENSION(0:n-1) :: vector
REAL(kind = 8), DIMENSION(1:m, n) :: matrix
CHARACTER(len=80), DIMENSION(24) :: screen
TYPE(my_own_type), DIMENSION(1:10) :: object
```

Array Declaration and Syntax

- Consider the following Matrix-Vector multiply as an example

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

Array Declaration and Syntax

- In older Fortran, arrays were traditionally accessed **element-by-element** -basis:

```
INTEGER, PARAMETER :: m = 4, n = 5
REAL(kind = 8) :: A(m,n), x(n), y(m)
INTEGER :: i, j

DO i = 1,m
  y(i) = 0
END DO

outer_loop: DO j = 1, n
  inner_loop: DO i = 1, m
    y(i) = y(i) + A(i , j) * x(j)
  END DO inner_loop
END DO outer_loop
```

Array Declaration and Syntax

- You can still do that but the **array syntax** potentially improves readability of your code
- It may also give the Fortran compiler a chance for better performance optimization

```
INTEGER, PARAMETER :: m = 4, n = 5
REAL(kind = 8) :: A(m,n) , x(n), y(m)
INTEGER :: j

! Array syntax requires less explicit DO-loops
y(:) = 0

loop: DO j = 1, n
  y(:) = y(:) + A(: , j) * x(j)
END DO loop
```

Array Initialization

- To make a program meaningful, we need to feed its variables with some values
- Arrays can be initialized **element-by-element**, **copied from another array**, or by using **array syntax** or **single line data initialization statements**
- More advanced initialization involves use of FORALL and WHERE -statements, or use of RESHAPE -intrinsic function

Array Initialization

- Element-by-element initialization:

```
DO j = 1, n
  idx(j) = j
  vector(j) = 0
END DO
```

- Initialization using array syntax:

```
vector = 0   ! or equivalently
vector(:) = 0
```

Array Initialization

- Initialization by copying from another array:

```
REAL(kind=8) :: to_matrix(100,100),from_matrix(0:199, 0:199)
to_matrix(1:100, 1:100) = from_matrix(0:199:2, 0:199:2)
! Every 2nd element from from_matrix is copied
```

- Using array construction and implied-DO:

```
INTEGER, PARAMETER :: fixed(2:4) = (/ 20, 30, 40 /)
INTEGER :: idx(0:10)
DATA idx / 0, 1, 2, 3, 7 * 0 /

! or

idx(0:10) = (/ 0, (i, i = 1, 3), (0, i = 4,10) /)
```

Array Sections

- With Fortran array syntax we can access a part of an array in an intuitive way

```
vector (first : last : increase )
```

- Array sections are perhaps the main reason for Fortran usability in scientific computing

```
Sub_Vector ( 3 : N + 8) = 0
Every_Third ( 1 : 3 * N + 1 : 3 ) = 1
Diag_Block ( i - 1 : i + 1, j - 2 : j + 2 ) = k
```

- Sections enable us to refer to a sub-block of a matrix or a sub-cube of a 3D-array:

```
REAL(kind = 8) :: A (1000, 1000)
INTEGER(kind = 2) :: pixel_3D(256, 256, 256)

A(2:500, 3:300:3) = 4.0_8 ! 8 assigns the precision for REAL
pixel_3D (128:150, 56:80, 1:256:8) = 32000
```

Array Sections

- Be aware of: when copying array sections, then both left and right hand sides of the assignment statement has to have conforming dimensions:

```
LHS(1:3, 0:9) = RHS(-2:0, 20:29)

!  but an error if

LHS(1:2, 0:9) = RHS(-2:0, 20:29)
```

Passing Array Arguments to Procedures

Three ways to pass argument to procedures:

- **Assumed shape array**

```
REAL, DIMENSION(:,:) :: matrix
```

- **Explicit shape array**

```
REAL, DIMENSION(size1,size2) :: matrix
```

- **Assumed size array**

```
REAL, DIMENSION(low:up,*) :: matrix
```

Assumed Shape Arrays

```
REAL, DIMENSION(:,:) :: matrix
```

- Type, kind and rank of formal argument must be the same as actual argument
- Extent of a dimension (shape) is same as in the actual argument array
- Must have an explicit interface

```
SUBROUTINE f(x,y,z)
  IMPLICIT NONE

  REAL, DIMENSION(:,:) :: x, y, z
  INTRINSIC SIN, SQRT

  z = SIN(SQRT(x**2 + y**2))

END SUBROUTINE f
```

```
PROGRAM  arrays
  IMPLICIT NONE

  INTEGER, PARAMETER :: n = 5
  REAL, DIMENSION(n) :: t
  REAL, DIMENSION(n,n) :: x, y, z
  INTEGER :: i

  t = (/ (10.0*i/(n-1)-5.0, i = 0, n-1) /)
  DO i = 1,n
    x(i,1:n) = t
    y(1:n,i) = t
  END DO

  CALL f(x,y,z)
  WRITE (*,*) z

  CONTAINS

    Declaration of SUBROUTINE f(x,y,z)

END PROGRAM  arrays
```

Explicit Shape Arrays

```
REAL, DIMENSION(size1,size2) ::  matrix
```

- Rank doesn't have to match the actual argument array
- Array bounds must be passed or be defined via module

```
SUBROUTINE sub_1(x,n)
  IMPLICIT NONE

  INTEGER :: n
  REAL, DIMENSION(n*n) :: x

  x(8:20) = -1.0

END SUBROUTINE sub_1
```

```
SUBROUTINE sub_3(x,n)
  IMPLICIT NONE

  INTEGER :: n
  REAL, DIMENSION(n,n/2,2) :: x

  x(3,:,:)  = 2.0

END SUBROUTINE sub_3
```

```
PROGRAM  explicit_shape
  IMPLICIT NONE

  INTEGER, PARAMETER :: n = 5
  REAL, DIMENSION(n,n) :: x = 0.0

  CALL sub_1(x,n)
  WRITE(*,'(5(F8.3))') x(3,:)

  CALL sub_3(x,n)
  WRITE(*,'(5(F8.3))') x(3,:)

  CONTAINS

    Declaration of SUBROUTINE sub_1(x,n)
    Declaration of SUBROUTINE sub_3(x,n)

END PROGRAM  explicit_shape
```

Output:

```
0.000   -1.000  -1.000  -1.000  0.000
2.000   2.000   2.000   2.000   0.000
```

Assumed Size Arrays

```
REAL, DIMENSION(low:up,*) ::  matrix
```

- Declaration specifies the rank and the extents for all dimensions except the last
- Dimensions may change when called
- Fortran 77 relic. Do not use!

Passing Array Sections to Procedures

- Also array sections — not necessarily full arrays — can be passed into a procedure :

```
INTEGER, DIMENSION(10,20) :: Array
CALL SUB(Array)
CALL SUB(Array(5:10, 10:20))
CALL SUB(Array(1:10:2, 1:1))
CALL SUB(Array(1:4, 1:))
CALL SUB(Array(:10, :))
```

- Note that an array section is usually copied into a hidden temporary array upon calling a procedure and copied back to the array section upon return
- This may have some unwanted side- effects (like array overwrite with incorrect values) when using shared memory based parallel processing, such as OpenMP

Array Valued Functions

- Result variable array
- Shape must be known at execution

```
FUNCTION f(x) RESULT(z)
  IMPLICIT NONE

  REAL, DIMENSION(:)   :: x
  REAL, DIMENSION(size(x)) :: z

  z = 2*x

END FUNCTION  f
```

Array Intrinsic Functions

- Fortran built-in array (intrinsic) functions enable operations on multiple array elements in one go
  - they can apply various operations on whole array, not just elements
- As a result either another array or just a scalar value is returned
- Subset selection through masking possible
  - Operations are performed only for those elements of array where corresponding elements of the logical array mask are .TRUE.
- Masking and use of array (intrinsic) functions is often accompanied with use of FORALL and WHERE array statements

Array Intrinsic Functions

- The most commonly used array functions are the following:
  - SIZE (array [, dim]) returns # of elements in the array [, along the specified dimension]
  - SHAPE (array) returns an INTEGER vector containing SIZE of array in each dimension
  - COUNT (larray [,dim]) returns count of elements which are .TRUE. in logical larray
  - SUM (array[, dim][, mask]) sum of the elements [, along dimension] [, under mask]
  - ANY (larray [, dim]) returns a scalar value of .TRUE. if any value in larray is .TRUE.
  - ALL (larray [, dim]) returns a scalar value of .TRUE. if all values in larray are .TRUE.
  - MINVAL (array [,dim] [, mask]) returns the minimum value in a given array [along specified dimension] [, under mask]
  - MAXVAL is the same as MINVAL, but returns the maximum value in a given array
  - MINLOC (array [, mask]) returns a vector of location(s) [, under mask], where the minimum value(s) is/are found
  - MAXLOC similar to MINLOC, for maximums
  - RESHAPE (array, shape) returns a reconstructed array with different shape than in the input array

Array Intrinsic Functions

- Some array functions manipulate vectors and matrices effectively:
  - DOT_PRODUCT (a_vec, b_vec) returns a scalar value — dot product — of two vectors
  - MATMUL (a_mat, b_mat) returns a matrix containing matrix multiply of two matrices
  - TRANSPOSE (a_mat) returns a transposed matrix of the input matrix

```
PROGRAM matrix
  IMPLICIT NONE

  INTEGER, PARAMETER :: n = 5
  CHARACTER(LEN=*), PARAMETER :: form = '(A,5F7.2)'
  REAL, DIMENSION(n,n) :: mat
  INTEGER :: i, j

  ! Initialization of mat with RESHAPE
  mat = RESHAPE( (/ ( (i-j, i = 1, n), j = 1, n) /), &
    (/ n, n /) )
  WRITE(*,form) 'Sum of the elements of matrix', SUM(mat)
  WRITE(*,form) 'Sum of the column elements of matrix', SUM(mat,1)
  WRITE(*,form) 'Sum of the row elements of matrix', SUM(mat,2)
END PROGRAM matrix
```

Array Control Statements

- Array control statements FORALL and WHERE are commonly used in the context of manipulating arrays

```
PROGRAM exa_where
  IMPLICIT NONE

  INTEGER, :: j, ix(5)

  ix(:) = (/ (j, j=1,SIZE(ix)) /)
  FORALL (j=1:SIZE(ix), ix(j) < 3) ix(j) = 0

  WHERE (ix == 0) ix = -9999

  WHERE (ix < 0)
    ix = -ix
  ELSEWHERE
    ix = 0
  END WHERE
  WRITE (*,*) ix

END PROGRAM exa_where
```

Dynamic Memory Allocation

- Sizing of arrays maybe static or dynamic
- Dynamic memory allocation provides effective runtime sizing of your data arrays
- For small array sizes a static dimensioning is usually not a problem
- For large arrays dynamic memory allocation is maybe the only option. Otherwise the program may not fit into the memory — and will not be able to run
- Variable declaration has an ALLOCATABLE (or a POINTER) attribute, and memory is allocated through ALLOCATE -statement
- A variable, which is declared in the procedure with size information coming from the argument list or a MODULE, is an **automatic array**: no ALLOCATE is needed

Dynamic Memory Allocation

- An example of using ALLOCATE:

```
INTEGER :: m, n, alloc_stat
INTEGER, ALLOCATABLE ::  idx(:)
REAL(kind = 8), ALLOCATABLE :: mat (: , :)
m = 100; n = 200
ALLOCATE (idx(0:m-1), STAT=alloc_stat)
IF (alloc_stat /= 0) CALL abort( ) ! Memory allocation failed
ALLOCATE ( mat(m,n) , STAT=alloc_stat)
IF (alloc_stat /= 0) CALL abort( )
DEALLOCATE (idx,mat)
```

- An identical example with automatic arrays:

```
SUBROUTINE sub(m)
  USE some_module, ONLY : n
  IMPLICIT NONE

  INTEGER, INTENT(in) :: m
  INTEGER :: idx(0:m-1)
  REAL(kind = 8) :: mat(m,n)
  ! No explicit ALLOCATE/DEALLOCATE!
  ...
END SUBROUTINE sub
```

Dynamic Memory Allocation

- When using ALLOCATE -statement, it is always recommended to use ALLOCATABLE rather than POINTER attribute in dynamic variable declaration
- To avoid unexpected memory growth ('memory leak'), please do remember to use DEALLOCATE for every ALLOCATE statement ever used
  - This has been relaxed now and some newer Fortran compilers will automatically de-allocate ALLOCATABLE (but not POINTER) arrays, when they go out of scope

Pointers to Arrays

- POINTER attribute enables a versatile alias mechanism to arrays and array sections (or even scalars)
- POINTER — if misused — too often leads to a hard-to-detect programming error
- Not considered in this course.

FORTRAN95:

INPUT/OUTPUT (I/O)

I/O Formatting

- To prettify the output and to make it human readable, use **FORMAT** descriptors in connection with WRITE -statement (can be used with READ -statement as well)
- This can be done either through FORMAT -statements, CHARACTER variable or embedded in READ / WRITE fmt -keyword

```
PROGRAM  print_out
  ! Print out an integer n to the field with width 5.
  IMPLICIT NONE

  INTEGER :: n = 34
  CHARACTER(LEN=*), PARAMETER :: form = '(I5)'

  WRITE(*, '(I5)') n ! fmt -keyword
  WRITE(*, form) n ! CHARACTER variable (parameter)
  WRITE(*, 30) n ! Using FORMAT -statement.  Do not use!
  30 FORMAT (I5)

END PROGRAM print_out
```

I/O Formatting

Basic data edit descriptors:

Data type	Basic data edit descriptors	Examples	
Integer	Iw, Iw.m	I5, I5.3	`WRITE(*,'(I5)') j` `WRITE(*,'(I5.3)') j`
Real (decimal and exponential forms)	Fw.d Ew.d, Ew.dEe	F7.4 E12.3, E12.3E4	`WRITE(*,'(F7.4)') r` `WRITE(*,'(E12.3E4)') r`
Character	A, Aw	A, A20	`WRITE(*,'(A)') c`
Logical	Lw	L7	`WRITE(*,'(L7)') t`

where

- w is the total width of the field (filled with *** if overflow)
- m is the least number of digits in the (sub)field (optional)
- d is the number of digits to the right of decimal point
- e is the number of digits in the exponent subfield

I/O Formatting: Examples

```
WRITE(*,'(I5)') 9                  ! Output:     9
WRITE(*,'(I5.3)') 9                ! Output:   009

WRITE(*,'(F7.2)') 12.3443          ! Output:   12.34
WRITE(*,'(F9.4)') 12.3443          ! Output:   12.3443
WRITE(*,'(F9.4)') 37283845.3872    ! Output: *********
WRITE(*,'(E12.3)') 12.3443         ! Output:   0.123E+02
WRITE(*,'(E12.5)') 1.243E-5        ! Output:  0.12430E-04
WRITE(*,'(E15.3E4)') 1.243E-5      ! Output:     0.124E-0004

WRITE(*,'(A4)') 'hel'              ! Output:  hel
WRITE(*,'(A4)') 'hello'            ! Output: hell
WRITE(*,'(A)') 'hello'             ! Output: hello
```

File I/O

Basic concepts:

- Writing to or reading from a file is basically similar to writing onto a terminal screen or reading from a keyboard
- Differences:
  - File must be opened first with OPEN-statement, in which the unit number and (optionally) a file name are given
  - Subsequent writes (or reads) must to refer to the given unit number
  - File should finally be closed
- For example:

```
INTEGER :: iu
iu = 12
OPEN(unit=iu, file = 'foo')
WRITE(iu, ...)  ! or READ(iu, ...)
CLOSE(iu)
```

File I/O: Opening and Closing a file

- The syntax for file opening and closing is:

```
OPEN([unit=] iu, file = 'name' [, options])
CLOSE([unit=] iu [, options])
```

  For example:

```
OPEN(10, file= 'output.dat', status='new')
CLOSE(unit=10)
```

- The first parameter is the unit number
- The keyword unit= can be omitted
- The unit numbers 0, 5 and 6 are predefined
  - 0 is output for standard (system) error messages
  - 5 is for standard (user) input
  - 6 is for standard (user) output

  These units are opened by default and should not be closed
- File opening options are not considered in this course.

File I/O: File Properties

- File properties can be obtained by using INQUIRE -statement.
- The syntax has two forms, one based on file name, the other for unit number

```
INQUIRE(file='name', options ...)
INQUIRE(unit=iu, options ...)
```

- The options contains one or more (keyword , variable) -pairs
- The corresponding variable contains the information that was inquired
- There are plenty of various keywords not considered in this course.
- An example: Find out about file existence

```
LOGICAL :: exfile
INQUIRE(file='foo.dat', exists=exfile)
IF (.NOT. exfile) THEN
  WRITE(*,*) 'The file does not exist'
ELSE
  ...
END IF
```

File I/O: Using READ and WRITE

- Reading from and writing to a file is done by giving the corresponding unit number (iu) as a parameter:

```
READ(iu,*) str
READ(unit=iu, fmt=*) str
WRITE(iu,*) str
WRITE(unit=iu, fmt=*) str
```

- Formats and other options can be used as needed
- The asterisk (*) format indicates list directed output (you do not choose the output style)
- If the keyword unit is used, also fmt -keyword must be used

File I/O: Using READ and WRITE

- If the file unit (iu) has not been explicitly OPENed, the very first READ or WRITE will trigger an implicit OPEN
  - In most UNIX systems this means opening a file named as 'fort.⟨iu⟩', where ⟨iu⟩ is the unit number in concern
- The default output and input units are referred with asterisk:

```
READ(*,...) str
WRITE(*,...) str
```

Note that they are NOT necessarily the same as the unit numbers 5 and 6 earlier

File I/O: Format Descriptors

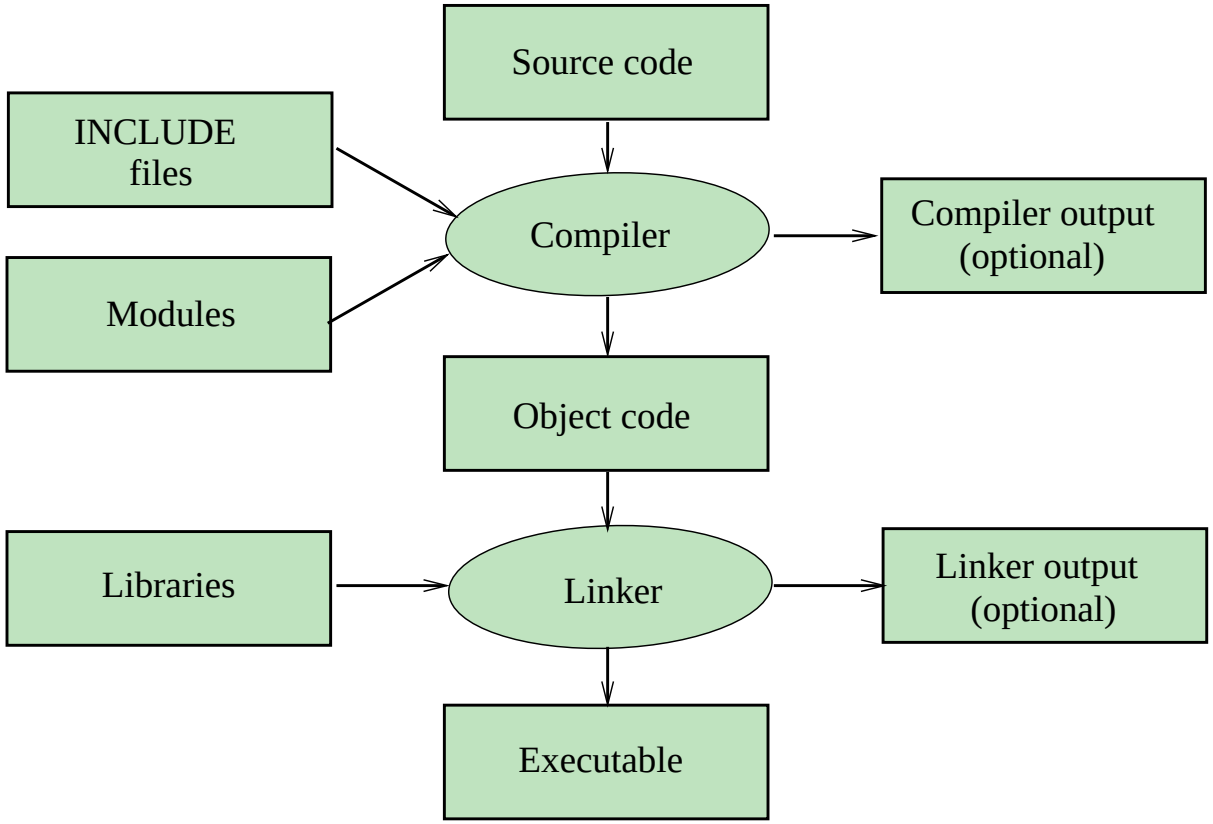
Control edit descriptors (n is the number of characters):

Descriptor	Task	Example
/	New line	`WRITE(*,'(I5,/,I5)') i, j`
nX	Number of blanks	`WRITE(*,'(I5,5X,I5)') i, j`
Tn	Tab to specified column	`WRITE(*,'(I5,T20,I5)') i, j`
TRn	Tab right n columns	`WRITE(*,'(I5,TR5,I5)') i, j`
TLn	Tab left n columns	`WRITE(*,'(I5,TL3,I5)') i, j`
BN	Do not read blanks	`READ(*,'(BN,I5)') i ! (default)`
BZ	If blanks → zeros	`READ(*,'(BZ,I5)') i`
SP	Switch on plus sign	`WRITE(*,'(SP,I5)') i`
SS	Switch off plus sign	`WRITE(*,'(SP,I5,SS,I5)') i, j`

Fortran95:

Compiling & Linking

Compiling and Linking



Compiling and Linking

- Suppose we have four Fortran source files as follows

`main.f95`	Main program, which calls `func1` and `sub1` defined in files `func1.f95` and `sub1.f95`, respectively. Both, `func1` and `sub1` are external to the `main` program.
`global.f95`	Module file containing several global variables. This module is used from the `main` program and from `func1`.
`func1.f95`	Contains one function, called: `func1`.
`sub1.f95`	Contains one subroutine, called: `sub1`.

- In order to build the main program, we need to compile all four source files to object files and link them together to produce the executable file. Using the `gfortran` following commands could be issued

```
gfortran exec_filename global.f95 main.f95 func1.f95 sub1.f95
```

- If you leave out the `exec_filename` the compiler produce an executable file called `a.out`.
- In our current directory we should find following additional files produced by the compiler: `global.o`, `global.mod`, `main.o`, `func1.o` and `sub1.o`.

Compiling and Linking

- In order to compile single source file without automatic linking, we should issue the following command:

```
gfortran -c sub1.f95
```

- To compile a source file which uses a module file (through the USE -statement) we need to compile the module file first. For example:

```
gfortran -c global.f95
gfortran -c main.f95
```

- Final linking of all compiled (module files) is carried out with the following command:

```
gfortran -o exec_filename global.o main.o func1.o sub1.o
```

or just by typing:

```
gfortran -o exec_filename *.o
```

Compiling and Linking: Example

Compiling and running the code:

```
% gfortran -c units.f95
% gfortran -o test_units test_units.f95 units.o
% ./test_units
inch to cm:  2.539999962
```

Note that in order to use the MODULE units in the main program, it has to be compiled before the main program.

```
MODULE units
  IMPLICIT NONE
  REAL, DIMENSION(6), PARAMETER :: &
    dist = (/ &
    0.01, & ! centimeter
    0.0254, & ! inch
    0.3048, & ! foot
    0.9144, & ! yard
    1e3, & ! kilometer
    1609.34 /) ! mile
END MODULE units
```

```
PROGRAM  test_units
  USE units
  IMPLICIT NONE

  WRITE (*,*) 'inch to cm: ', dist(2)/dist(1)

END PROGRAM  test_units
```

Unix Programming Environment:

Introduction & Motivation

Unix Programming Environment

- Unix system provides many tools for program development
- In what follows we are going to familiarize ourselves with two of them:
  - Program compilation and linking: `make`
  - Program debugging: GNU debugger `gdb`
- The other quite useful tools are e.g. `gprof`, `awk`, `sed`, `gnuplot` and `cvs`.
  - Type `man command_name` to get more information of these tools.

Unix Programming Environment:

Make & Makefile

UNIX PROGRAMMING ENVIRONMENT: MAKE

- Say you have a code that contains tens of source code files.
- Quite often e.g. the modules used by the program are all in separate files.
- Every time one of the source files change, you need to compile it and link it against all other object files necessary to produce the executable file. However, when you modify only one file it is useless to compile all source files
  - Only those files need to be compiled that depend on the modified one.
  - The `make` system automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
  - `Make` gets its knowledge of how to build your program from a file called `makefile`. When you write a program, you should write a `makefile` for it, so that it is possible to use `make` to build and install the program.
- Our examples show Fortran 95 programs, but you can use `make` with any programming language (for example: C or C++) whose compiler can be run with a shell command.

MAKE: A SAMPLE MAKEFILE

```
# This is an commentary line in a makefile
# Start of the makefile

exec_filename:  global.o main.o func1.o sub1.o
   gfortran -o exec_filename global.o func1.o sub1.o main.o

global.mod:  global.o global.f95
   gfortran -c global.f95 # Note, these must be tabs not spaces

global.o:  global.f95
   gfortran -c global.f95

main.o:  global.mod main.f95
   gfortran -c main.f95

func1.o:  global.mod func1.f95
   gfortran -c func1.f95

sub1.o:  sub1.f95
   gfortran -c sub1.f95

clean:
   rm global.mod global.o main.o func1.o sub1.o exec_filename

# End of the makefile
```

Make and Makefile

- This file should be saved under the directory where the source files are, and it should be given a name makefile.
- To use makefile to create the executable file called `exec_filename`, type:

```
make
```

- To use makefile to delete the executable file and all the object files from the directory, type:

```
make clean
```

MAKE AND MAKEFILE

- A simple `makefile` contains *rules* that are of the form:

```
targets:  prerequisites
    command1
    command2
    ...
```

- Please Note: There is a *tab* character before the `command`, not *spaces*.
- A `target` is usually the name of a file that is generated by a program; examples of targets are executable or object files.
- A `prerequisite` is a file that is used as input to create the `target`.
- A `command` is an action that `make` carries out. A rule may have more than one `command`, each on its own line.
- A `target` can also be the name of an action to carry out, such as `clean`, and it need not have prerequisites.

Make and Macros

- You can define variables (they are also called macros) in `makefile`:

```
OBJ = global.o main.o func1.o sub1.o
```

- Then use this variable when appropriate, as follows:

```
exec_filename:  $(OBJ)
    gfortran -o exec_filename $(OBJ)

clean:
    rm exec_filename $(OBJ)
```

- You can also define the name of the used compiler as a macro:

```
FC = gfortran
```

- And you can define compiler options (e.g. optimization level) as a macro:

```
FFLAGS = -O3
```

MAKE: A SAMPLE MAKEFILE (VERSION 2)

```
# Start of the makefile
# Defining variables
OBJ = global.o main.o func1.o sub1.o
FC = gfortran
FFLAGS = -O3

exec_filename:  $(OBJ)
    $(FC) -o exec_filename $(FFLAGS) $(OBJ)

global.mod:  global.o global.f95
    $(FC) -c $(FFLAGS) global.f95

global.o:  global.f95
    $(FC) -c $(FFLAGS) global.f95

main.o:  global.mod main.f95
    $(FC) -c $(FFLAGS) main.f95

func1.o:  global.mod func1.f95
    $(FC) -c $(FFLAGS) func1.f95

sub1.o:  sub1.f95
    $(FC) -c $(FFLAGS) sub1.f95

clean:  # Cleaning everything
    rm global.mod exec_filename
    rm $(OBJ)
# End of the makefile
```

Unix Programming Environment:

Program Debugging

UNIX PROGRAMMING ENVIRONMENT: DEBUGGING THE CODE

- A common way to get information on the program behavior is the so called technical student's debugger: scatter print (write) commands around the code.
  - This can be quite tedious and has also problems with e.g. buffering of the output.
- For more versatile debugging you have to use a real debugger program, which allows you to:
  - Stop the program execution, examine the program state (variables etc.), and continue execution.
  - Stop program execution when a certain source line or routine is entered (breakpoints).
  - Print out variable values.
  - Change variable values.
  - Be informed when the value of a certain variable is changed.
  - Run the code one source line at a time.
  - Print the subroutine call stack.
  - Investigate a core dump.
- In Linux systems this is called `gdb` (GNU Debugger).

gdb: Compiling the program

- In order to use a debugger like `gdb` to track the execution of your program, it is necessary to compile the program with the `-g` option.
- It is also a good idea to turn off optimization: `-O0`
- For example, suppose your source file is called `foo.f95`. To compile it, type:

```
gfortran -O0 -g -o foo foo.f95
```

- The command above will create an executable file named `foo`. You can run this executable normally, but you can also run it under the control of the `gdb` debugger to find out what it is actually doing.

gdb: Starting Your Program with gdb

- To start execution of a program named `foo` under `gdb`:
  - Type `gdb` followed by the name of your program, for example:

    ```
    gdb foo
    ```

    You will get a command prompt that looks like this: (`gdb`)
  - Type these commands at the (`gdb`) prompt:

    ```
    break main
    run
    ```

  - This will start execution of your program, but execution will pause just before the first executable statement.

gdb: Stepping Through Your Program

- One thing that is good to know is the exact sequence of execution of your program, especially through loops and conditional branches.
- If the program is not too large, you can follow it easily by executing one line at a time.
- To execute the next statement, type:

```
step
```

- Each time you type a `step` command, `gdb` will list the line that it is about to execute, with the line number on the left. So you can see what is about to happen before it happens.

gdb: Finding Out Where You Are

- To find out where you are at any time, type the command:

```
where
```

  This will show you the current line number. For example, a line like this:

```
#0 foo () at foo.f95:12
```

  shows that the execution of our program is currently at a location that corresponds to line 12 in the source file `foo.f95`.

- You can display a few lines of your source program around the current location by using the command:

```
list
```

- You can also specify a range of lines to be listed. For example, to list lines 10 through 24 in the current program, type:

```
list 10,24
```

gdb: Displaying the value of variable

- At any time while you are stepping through the execution of your program, you can find out what values are currently stored in your variables by using the `print` command.
- For example, if you have a variable named `density`, you can type this command to examine its value:

```
print density
```

- Warning: You must type your variable names all in lowercase letters in `gdb`, regardless of how they were capitalized in your source program.

gdb: Terminating Your Program

- To exit `gdb`, type this command:

```
quit
```

You will get this message:

```
The program is running.  Quit anyway (and kill it)?  (y or n)
```

Type `y` to confirm that you want to exit.

gdb: Most Common Commands

Command	Description	Example
`help`	Help.	
`run`	Start the program. Command line arguments can also be given.	`run arg1 arg2`
`list`	List source.	`list sub1`
`print`	Print expression.	`print 10*y(1)`
`where`	Print procedure call stack.	`where`
`break`	Set breakpoint.	`break sub1` `break sub2.f95:7`
`continue`	Continue run.	
`step`	Step program until it reaches a different source line.	
`next`	Step program, proceeding through subroutine calls.	
`set`	Set the value of a variable.	`set y(5)=10.0`
`display`	Print out the value of a variable after every gdb command.	`display y`
`watch`	Track changes of a variable.	`watch x`
⟨`control`⟩`-C`	Stop the program execution. Continue with commands `continue`, `step` or `next`.	
⟨`return`⟩	Repeat the previous command.	

References and Further Information

- **Fortran 95/2003 Explained** by Michael Metcalf, John Reid and Malcom Cohen, Oxford University Press 2004.
- **Fortran 95/2003** (in Finnish, main source for this course material) by Juha Haataja, Jussi Rahola and Juha Ruokolainen, CSC — Tieteellinen laskenta Oy., 2007.
- **Fortran 95 -course at CSC** by Tommi Bergman, Jarmo Pirhonen and Sami Saarinen, 2010 (main source of the exercise material used in this course):

  http://www.csc.fi/english/csc/courses/archive/fortran2010
- **Tools for High Performance Computing 2009** -course material by Antti Kuronen, 2009

References and Further Information

- **Fortran Wiki**:
  http://fortranwiki.org/fortran/show/HomePage
- **gfortran** — the GNU Fortran compiler home page:
  http://gcc.gnu.org/wiki/GFortran
- **Fortran libraries**:
  http://en.wikipedia.org/wiki/List_of_numerical_libraries#Fortran
- **Makefile learning tutorial for Fortran**:
  http://genius2k.is-programmer.com/posts/40301.html
- **Debugging Fortran programs with gdb**:
  http://infohost.nmt.edu/tcc/help/lang/fortran/gdb.html