united nations
educational, scientific
and cultural
organization

international atomic
energy agency

the
abdus salam
international centre for theoretical physics

SMR.1492 - 1

Winter College on
NUMERICAL METHODS IN ELECTRONIC STRUCTURE THEORY

20 January – 4 February 2003

ADVANCED PROGRAMMING TECHNIQUES
(Fortran 90)

Carlo CAVAZZONI
CINECA, Bologna, ITALY

CINECA

Advanced Programming Techniques
(Fortran 90)

Carlo Cavazzoni
(High Performance Computing Group)
CINECA

CINECA

Outline

Introduction to FORTRAN90
Syntax Elements
Intrinsic Type
Basic Programming Constructs
Array, Dynamic Allocation
Pointers
Data Types
Program Units
Modules vs Common
Modular Programming vs Procedural Programming
Data Abstraction, Data Hiding

CINECA

Why Fortran90

Fortran is the language that provide the best syntax constructs for scintific and technical program (intrinsic complex, intrinsic math functions, array syntax)

Fortran is less flexible (compared to other Languages, C and C++) in memory management, this allow compilers to produce executables which are more efficient especially in floating point operations.

The learning curve of Fortran is by fare more steep than that of C and C++, you need to know less about computer programming

CINECA

Basic Syntax Elements

Contrary to F77, F90 default source code layout is FREE
All columns are equivalent (you can start writing in the first col.)
The default lenght of a line is **132** characters.

To continue a statement in a new line you shoud use the **&** character at the end of the line and optionally at the beginning of the new line.

Comment always begins with the **!** character.

You can place many F90 statements on a single line, using the **;** character as separator.

To make codes more readable is possible to separate declarations from variables name using **::** .

(program example: f90syntax.f90)

CINECA

F90 Intrinsic Types

F90 Intrinsic type are the same as F77:

integer, real, complex, logical, character

Each intrinsic type is associated to a default kind that define the range of values that could be represented
(i.e. all integer between -128 and +127)

The programmer could change the kind (therefore the range) of the intrinsic type.

Examples

```
integer, parameter :: i6 = selected_int_kind(6)
! this select the integer kind whose range is at lest -999999, 999999
integer(i6) :: m
! this declare m as an integer of kind=i6
```

(program example: kind1.f90 and kind2.f90)

CINECA

Basic Programming Constructs

Inherited from F77:

IF ... THEN ... ELSE ... END IF
DO ... END DO

New:

SELCET CASE ... CASE ... CASE DEFAULT ... END SELECT
DO WHILE ... END DO

(program example: control1.f90 and control2.f90)

New I/O Feature – Namelist I

Namelist let you group a set of variables to facilitate the input output operation on the group as a whole.

```
integer            :: atomic_number
real               :: atomic_mass
character(len=2) :: atom_name
namelist / atom / atomic_number, mass, atomic_weight, atom_name

.
.

read(5, nml = atom )
.
.
```

(program example: f90namelist.f90)

New I/O Feature – Namelist II

The input file will look like:

```
&atom
  atom_name = 'Si',
  atomic_weight = 28.086,
  atomic_number = 14
/
```

(program example: f90namelist.inp)

Array I

The array features of F90 and the array intrinsic procedures, are one of the big new issue of the language.

Standard Array Declaration

```
integer, parameter :: n = 10
real :: a(n,n)
```

this work if the size of the array is know at compile time

Allocatable Array Declaration

```
real, allocatable :: a(:,:)
        ! read n somewhere else
allocate(a(n,n))    ! associate a with fisical memory
        ! do the job with a
deallocate(a)               ! free the memory
```

this is by fare more flexible, and let you save lot of memory

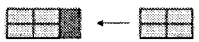
Array II

It is possible to assign a value to a set of array elements

```
allocate( a(n,n) )
a = 0.0                          ! clear the array
a( 1:n/2, n/2:n ) = 1.0          ! set an array section
```

array copy

```
allocate( b(n,m) )          ! m > n
b(1:n, 1:n) = a             ! copy a in a section of b
```



Array III

Array Algebra

```
allocate( b(n,m), a(n,n), c(m,n) )  ! m > n

        ! set a and c

b(1:n, 1:n) = a + c(1:n, : ) * 2.0  ! is functionally
                                    ! equivalent to a loop

c(1:n, : ) = sqrt( a )          ! assign to c the square root of a
                                ! element by element
```

Any Intrinsic Functions could be coupled with array expression, they act on an element by element base.

Array IV

There are NEW array intrinsic functions

```
allocate( b(n,m), a(n,n), c(m,n), d(m,n,l)  ! m > n
        ! set b, c and d
a = matmul( b, c )  !  matrix multiplication
                        ! ( only for rank 2 array )
b = transpose( c )  !  matrix transpose
                        ! ( only for rank 2 array )

a = matmul( b, d(:,:,l) )       !  l<l, work with 2D sub array

allocate( x(n), y(n) )
        ! set x and y
dp = dot_prosuct( x, y ) !  dot product ( only for rank 1 array )

s  = sum( b )                   !  sum of the element of b
bmax = maxval( b )!  maximum among b element
bmin  = minval( b ) !  minimum
```

Array V

Find out status, array size and shape

```
if( allocated( a ) ) then
  asiz = size( a )              ! number of element of a
  arow = size( a, 1 ) ! number of rows
  acol  = size( a, 2 ) ! number of columns
else
  asiz = 0
  arow = 0
  acol = 0
end if
```


Array VI

Logical array expression (mask)

A logical expression applied to array is called mask, If a is an array (a(n,n)) then a < 0.0 is a mask. Masks are used together with many intrinsic new functions.

```
nel = count( a < 0.0 )          ! number of negative elements

if( any( a > 0.0 ) ) then
  write(6,*) ' array a contains at least one element greather than 0 '
end if

if( all( a > 0.0 ) ) then
  write(6,*) ' array a contains only positive element '
end if
```

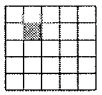

Array VII

There is a specific f90 constructs to manage mask

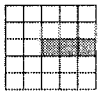
```
allocate( a(n,n), b(n,n) )

        ! set a

where ( a < 0.0 ) a = 0.0   ! clear all element less than zero

where ( abs( a ) > thr )
        b = 1 / a   ! get the inverse if the element
                    ! is less than the threshold
elsewhere
        b = 1 / thr! use for b the inverse of the threshold
end where
```


Array Examples

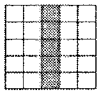
real :: ra(5,5)



ra(2,2)
ra(2:2:1, 2:2:1)

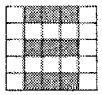


ra(3,3:5) shape(/3/)

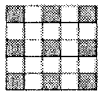


ra(:,3) shape(/5/)

Array Examples



ra(1::2,2:4) shape(/3,3/)



ra(1::2,1::2) shape(/3,3/)

Array Remarks

Remember all expression involving array work on an element by element base, and are functionally equivalent to a loop.

Only few intrinsic function, like matmul, transpose and dot_product involve all elements

Pointers

Usually, in other languages, like C, pointers refer to variables containing memory locations (they point to some memory location).
In F90 pointers are munch more than memory locations, they could be called memory descriptors, because they include not only the memory locations but also the description of the memory shape (number of dims, array boundaries, ecc...).

They are dangerously similar to allocatable array.

A FORTRAN90 pointer could be though as an empty container but with a well defined shape!!

The container could be filled with different contents at different moment in time, but compatible with the shape of the container.

Pointers II

```
Real, pointer :: a(:,:)
Real, targhet :: b(10,20)
Real, pointer :: c(:,:)

a => b     ! Now a and b are the same object
Allocate(c(100,200))
a => c     ! Now a and c are the same object
Nullify(a) ! Now a is empty
Deallocate(c)
a => b(1:5,10:20) ! Now a is a subblock b
```


Pointers III

```
Real, pointer :: ptr(:,:)
Real, targhet :: b(10,20), c(10,20)
Logical :: flg, mytest
Call init(b,c)
flg = mytest(....)
if(flg) then
  ptr => b
Else
  ptr => c
End if
... Now use ptr in the source code ...
```


Pointers IV

Pointer can be used in the same way as allocatable array

All we have said about allocatable array applay also to pointer, except some subtleties (i.e. a pointer that is not allocated or associated has an undefined state)

```
real, pointer :: a(:,:)
nullify( a ) ! makes a point to null
allocate( a(n,n) )
! use a like if it was an allocatable array
deallocate( a )
```

Pointer are useful in F90 as dynamic derevid type components, because in this context allocatable array are not allowed

Derived Data Types

- a derived data type is a type defined by the programmer
- are similar to the C structures
- are made up of intrinsic data types grouped together

```
type atom
  integer :: atomic_number
  real :: mass
  character(len=2) :: name
end type
```

derived data type allow to implement data abstraction

Derived Data Types II

derived type variable declaration

```
type (atom) :: litium
```

derived type assignement

```
litium = atom( 3, 6.94, 'Li' )
```

access to the type member (component selector)

```
write(6,*) 'the mass of', litium%name, 'is', litium%mass
```

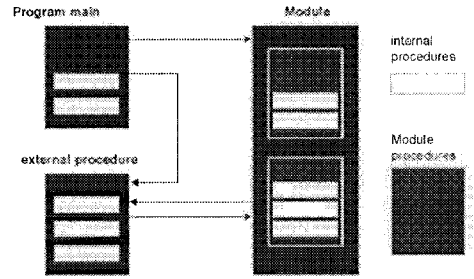
(program example: types1.f90)

Fortran Program Units

Program Units

F77:	F90:
PROGRAM SUBROUTINE FUNCTION	PROGRAM SUBROUTINE FUNCTION MODULE MODULE SUBROUTINE MODULE FUNCTION INTERNAL SUBROUTINE INTERNAL FUNCTION

Fortran90 Program units



Main Program

- Layout

```
PROGRAM [name]

        [specification statements]
        [executable statements]
        ........
END PROGRAM [name]
```

- Example

```
PROGRAM test
        ....
        ....
END PROGRAM test
```


Functions and Subroutines

Procedures are classified as:

- External – self-contained (not necessarely Fortran)
- Internal – inside a program unit
- Module – contained in a module

Fortran 77 allows only external procedures

(program example: internal.f90)

Layout:

```
SUBROUTINE name (dummy_argument_list)
        [specification statements]
        [executable statements]
        .....
END [SUBROUTINE [name]]

FUNCTION name (dummy_argument_list)
        [specification statements]
        [executable statements]
        .....
END [FUNCTION [name]]
```

Internal Procedures

- All program units could contain internal procedures
- Internal procedures are inserted below all other statements after the CONTAINS
- They have the same Layout of the external procedures, but require the statement END SUBROUTINE/FUNCTION
- The variables defined in the program unit are defined in the internal procedures too, a part when they are redefined in the internal procedure
- Nasted internal procedures are not allowed

Example

```
PROGRAM main
        IMPLICIT NONE
        REAL :: a, b, c
        REAL :: mainsum
        ....
        ....
        mainsum = add()
        ....
CONTAINS
  FUNCTION add()
    REAL :: add   ! a,b,c, defined in 'main'
    add = a + b + c
  END FUNCTION add
END PROGRAM main
```


Modules

- The modules allows to program using a modular paradigm
- It's a generalisation of the common block, used to share both data and program units
- Useful to define derived type and function that act on them (methods)
- Layout:

```
MODULE module_name
  [specification_statements]
  [executable_statements]
[CONTAINS
    module procedures]
END [MODULE [module_name]]
```

- the are accessed through the istruction USE

Modules: Global Data

- The variables defined in a module are global if the module is used in the main program or if the save attribute is used.
- If the SAVE attribute is used the module substitute the COMMON

```
MODULE globals
  REAL, SAVE ::a,b,c
  INTEGER, SAVE ::i,j,k]
END MODULE globals
```

- USE instruction: Examples

```
USE globals
! allow to access all the module variables
USE globals, ONLY :a,c
! allow the access anly of the two variablesi a and c
USE globals, r=>a, s=>b
! the variables a and b are accessed through the names r and s
```


Modules: Public and Private objects

- The program unit that includes a module through USE statement, could access all the module objects
- The access to the objects could be restricted by the use of the PRIVATE instruction

```
INTEGER, PRIVATE :: keep, out
```

(program example: module1.f90)

Modules: Module Procedure

- Modules could contains procedures that could be accessible to other program units
- Same Layout of the internal procedures:
  - the procedures should follow the instruction CONTAINS
  - the procedures should be ended with the END SUBROUTINE or END FUNCTION instruction
- They are especially useful to collect data types and function that act on the data types

Modules: Modules Procedures

```
MODULE point_module
 TYPE point
  REAL :: x,y
 END TYPE point
CONTAINS
 FUNCTION addpoints(p,q)
  TYPE(point), INTENT(IN)::p,q
  TYPE(point)::addpoints
  addpoints%x = p%x + q%x
  addpoints%y = p%y + q%y
 END FUNCTION addpoints
END MODULE point_module
```

```
USE point_module
TYPE (point):: px, py, pz
  .....
pz = addpoints(px,py)
```


Module vs Common

Module are often used in the place of common

(program example: com2mod.f)

(program example: com2mod.f90)

Module are by far more safe than common
(less error prone)

INTERFACE block: Explicit vs Implicit

- If an 'explicit' interface is used for a given procedure, the compiler could check for argument consistency
- Modules and Internal procedures have an 'explicit' interface by default
- External procedures have an 'implicit' interface by default
- *To specify an explicit interface for an external procedure, the INTERFACE* block should be used
- It is suggested to always use the block INTERFACE for the external procedures

Interfaces allow prototyping in F90, in a way similar to C

INTERFACE block: Layout

- Layout:

```
INTERFACE
 Interface_Body
END INTERFACE
```

Interface_Body it is an exact copy of the subprogram specifications.

- Example:

```
INTERFACE
 REAL FUNCTION func(x)
  REAL, INTENT(IN) :: x
 END FUNCTION func
END INTERFACE
```

(program example: interface0.f90)

Generic Procedures

- in Fortran 90 programmer defined generic functions are allowed
- The use of INTERFACE allow to link the correct procedure depending on the arguments:

```
INTERFACE geneeric-name
 specific_interface_body
 specific_interface_body
 .....
END INTERFACE
```

- A generic procedure could be called with the generic name, using the information of the arguments types the compiler will link the proper procedure
- The generic procedures with arguments of derived types are allowed only in the Modules

Interfaces allow operator and function overloading

Generic Procedures : Example

Two different external subroutines for REAL and INTEGER:

```
SUBROUTINE swap_real(a,b)
 IMPLICIT NONE
 REAL, INTENT(INOUT):: a,b
 REAL:: temp
 temp = a, a = b; b = temp
END SUBROUTINE swap_real
```

```
SUBROUTINE swap_int(a,b)
 IMPLICIT NONE
 INTEGER, INTENT(INOUT):: a,b
 INTEGER:: temp
 temp = a, a = b; b = temp
END SUBROUTIE swap_int
```

- Generic interface:

```
INTERFACE swap   ! generic name
 SUBROUTINE swap_real(a,b)
  REAL, INTENT(INOUT):: a,b
 END SUBROUTINE swap_real
 SUBROUTINE swap_int(a,b)
  INTEGER, INTENT(INOUT):: a,b
 END SUBROUTIE swap_int
END INTERFACE
```

Callig

CALL swap(x,y)

(program example: interface.f90)

Modules: Generic procedures

- Modules allow the use of generic procedures for derived type

(program example: interface2.f90)

Scope

Scoping Unit

- The scope of an entity is the ensamble of "*scoping units*" not overlapping in which this variable could be accessed without ambiguity

- Scoping unit:
  - definition of derived types
  - interface or procedure body
  - program unit or internal procedure

Scope / 1

Labels and names

- the **scope** of a label is the main program or a procedure
- Entities defined in different scoping units are different
- Inside a scoping unit, every entity should have a disctinct name
- The name of program units are global
- The scope of an entity defined in a module extend to the program units using (USE) that module

Scope: Example

```
MODULE scope1                          ! scope 1
...                                    ! scope 1
CONTAINS                               ! scope 1
  SUBROUTINE scope2                    ! scope 2
    TYPE scope3                        ! scope 3
      ...                              ! scope 3
    END TYPE                           ! scope 3
    INTERFACE                          ! scope 3
      ...                              ! scope 4
    END INTERFACE                      ! scope 3
    REAL x, y                          ! scope 2
100  ...                               ! scope 2
  CONTAINS                             ! scope 2
    FUNCTION scope5(...)               ! scope 5
      REAL y                           ! scope 5
      y = x + 1.0                      ! scope 5
100   ...                              ! scope 5
    END FUNCTION Scope5                ! scope 5
  END SUBROUTINE scope2                ! scope 2
END MODULE scope1                      ! scope 1
```


Procedure Arguments: INTENT

- Intent of the argument – specify if an argument is:
  - input (IN)
  - output (OUT)
  - both (INOUT)

- Examples:

```
INTEGER, INTENT(IN) :: in_only
REAL, INTENT(OUT) :: out_only
INTEGER, INTENT(INOUT) :: both_in_out
```


Procedures Arguments: Keywords

- Example

```
REAL FUNCTION area(star, finish, tol)
 IMPLICIT NONE
 REAL, INTENT(IN) :: start, finish, tol
 ....
END FUNCTION area
```

- Calls

```
a = area(0.0, 100.0, 0.01)
b = area(star=0.0, tol=0.01, finish=100.0)
c = area (0.0, finish=100.0, tol=0.01)
```

- If the keyword is used for a parameter it should be used for all the remaining parameters

Procedures Arguments: Optional

- Example

```
REAL FUNCTION area(star, finish, tol)
 IMPLICIT NONE
 REAL, INTENT(IN), OPTIONAL :: &
    start, finish, tol
   ....
END FUNCTION area
```

- Calls

```
a = area(0.0, 100.0, 0.01)
b = area(star=0.0, finish=100.0, tol=0.01)
c = area (0.0)
d = area (0.0, tol=0.01)
```


Procedures Arguments: PRESENT

- Example

```
REAL FUNCTION area(star, finish, tol)
 IMPLICIT NONE
 REAL, INTENT(IN), OPTIONAL :: &
    start, finish, tol
 REAL :: ttol
 ....
 IF (PRESENT(tol)) THEN
   ttol = tol
 ELSE
   ttol = 0.01
 ENDIF
 ....
END FUNCTION area
```

- The intrinsic function PRESENT allows to check if an argument is present.

Procedures Arguments : Derived Types

- The arguments of procedures could be of derived type if :
  - The procedure is internal to the program in which the type is defined
  - The derived type is defined in a Module used by the procedure

Procedures as Arguments

- In Fortran 77, a procedure arguments is always external
- In Fortran 90, the procedure passed as an argument should be an external or a module procedure.
- The internal procedures are not allowed as arguments

Procedures as Arguments /1

- Example

Calling program

```
.....
INTERFACE
  REAL FUNCTION func(x,y)
    REAL, INTENT(IN) :: x,y
  END FUNCTION func
END INTERFACE
.....
CALL area(func, start, finish, tol)
```

the passed function:

```
REAL FUNCTION func(x,y)
IMPLICIT NONE
  REAL, INTENT(IN) :: x,y
.....
END FUNCTION func
```


Functions: Clause RESULT

- The function could have the clause RESULT:

```
FUNCTION add(a,b,c) RESULT(sum_abc)
  IMPLICIT NONE
  REAL, INTENT(IN) :: a,b,c
  REAL :: sum_abc
   sum_abc = a+b+c
END FUNCTION add
```

- The clause RESULT is required for recursive functions

Array-valued Functions

- In Fortran 90 the functions could have an array as result:

```
FUNCTION add_vec(a, b, n)
 IMPLICIT NONE
 INTEGER, INTENT(IN):: n
 REAL, DIMENSION(n), INTENT(IN):: a, b
 REAL, DIMENSION(n):: add_vec
 INTEGER:: i
DO i = 1, n
    add_vec(i) = a(i) + b(i)
END DO
 END FUNCTION add_vec
```


Recursive Procedures

- There are two types of recurrence:
  - indirect: α call β that call α again
  - direct: α call itself (is required the clause RESULT)

```
RECURSIVE FUNCTION fact(n) RESULT(res)
 IMPLICIT NONE
 INTEGER, INTENT(IN):: n
 INTEGER:: res
 IF (n == 1) THEN
  res = 1
 ELSE
  res = n*fact(n-1)
 END IF
END FUNCTION fact
```


Program Structure

PROGRAM, FUNCION, SUBROUTINE, MODULE		
USE		
FORMAT	IMPLICIT NONE	
	PARAMETER	
	PARAMETER , DATA	TYPE, INTERFACE, variables declaration
	executable instructions	
CONTAINS		
Internal procedures		
END		

Program Structure

Use of INTERFACE blocks

- external procedure or module procedure:
  - operator or assignment overloading
  - generic name
- external procedure:
  - there are keyword or optional arguments
  - the procedure is a pointer o array-valued procedure
  - there are assumed-shape array or pointer/target arguments

Modular Programmin in FORTRAN 90

Procedural Programming

Decide which procedures you want;
use the best algorithms you can find

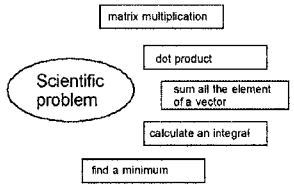
programming stile supported by FORTRAN77

Procedural Programming (F77)

- focus on procedures

divide the problem in a sequence of elementary operation

implement the best algorithms you can find



- no explicit dependencies among data
- no safe data grouping
- no data hiding

Modular Programming

Decide which modules you want; partition the program so that data is hidden within modules

programming stile supported by FORTRAN90

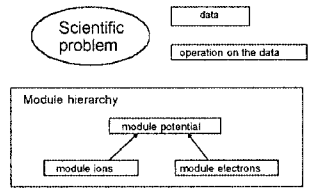
The MODULE *program unit makes possible the modular programming style*

Modular Programming

analize relations and dependencies among data

put related data in the same module togheter with the methods

build a module hierarchy



- modules are black boxes
- severe inflexibility

Object Oriented Programming

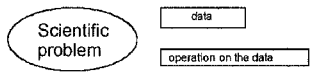
Decide which classes you want; provide a full set of operations for each class make commonality explicit by using inheritance

programming stile supported by C++

Object Oriented Programming

analize relations and dependencies among data

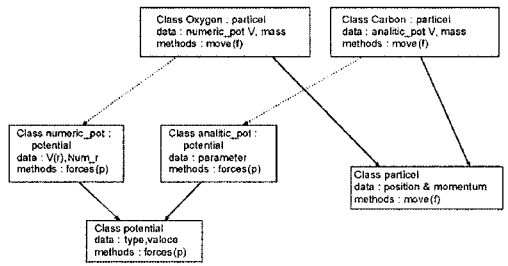
build objects, use inheritance to organize them



Add new object using the inheritance mechanism

- flexibility
- problems with efficiency

Object hierarchy



FORTRAN90 MODULE (sintax and properties)

```
MODULE derived
 USE base, ONLY: b
 REAL, PRIVATE :: a
CONTAINS
 SUBROUTINE set_value(v)
  REAL :: v
  a = v
 END SUBROUTINE set_value
END MODULE derived
```

explicit dependency

Data hiding

operation on data (methods)

Exemple 1 (module)

```
MODULE BLACK_BODY

 REAL*8, PRIVATE :: INTERNAL_TEMPERATURE = 0.0D0

CONTAINS

 SUBROUTINE HEATUP(D)
  REAL*8 D
  INTERNAL_TEMPERATURE = INTERNAL_TEMPERATURE + D
 END SUBROUTINE HEATUP

 SUBROUTINE COOLDOWN(D)
  REAL*8 D
  INTERNAL_TEMPERATURE = INTERNAL_TEMPERATURE - D
 END SUBROUTINE COOLDOWN

 REAL*8 FUNCTION TEMPERATURE()
  TEMPERATURE = INTERNAL_TEMPERATURE
 END FUNCTION TEMPERATURE

END MODULE BLACK_BODY
```


Esempio 1 (program)

```
PROGRAM EXEMPLE1

        USE BLACK_BODY
        REAL*8 DEGREE

        DEGREE = 100.0D0
CALL HEATUP(DEGREE)
WRITE(6,10) TEMPERATURE()
DEGREE = 50.0D0
CALL COOLDOWN(DEGREE)
WRITE(6,10) TEMPERATURE()

10 FORMAT(' BLACK BODY TEMPERATURE = ',F8.3)

END PROGRAM EXEMPLE1
```


Overloading

- assign to a symbol more than one meaning.
- use the same name for two different subroutine.
- the compiler can resolve conflicts using the contex.

Two kind of Overloading

- Overloading of functions or subroutines written by the programmer. Es: ***subroutine norma2*** (also known as generic procedures or functions)
- Overloading of intrinsic operators and functions. Es: ***+, *, mod ...***

Overloading Operators

- Modules allow the overloading of the intrinsic operator, to extend their use to derived data types (Operator overloading)
- this is optained with an INTERFACE block:

```
INTERFACE OPERATOR(intrinsic_operator)
        interface_body
END INTERFACE
```

Overloading Operators: Example

```
MODULE operator_overloading
 IMPLICIT NONE
 ....
 INTERFACE OPERATOR(+)
   MODULE PROCEDURE concat
 END INTERFACE
 ....
CONTAINS
 FUNCTION concat(cha,chb)
   CHARACTER(LEN=*), INTENT(IN)::cha, chb
   CHARACTER(LEN=LEN_TRIM(cha) + LEN_TRIM(chb))::concat
   concat = TRIM(cha)//TRIM(chb)
 END FUNCTION concat
 ....
END MODULE operatior_Overloading
```


Operator Definition

- The programmer could define new operators
- The operators name should be delimited by the character '.'
- The function should have one or two non optional arguments without INTENT(IN) specification
- Example: calculate the distance between two points

```
PROGRAM main
 IMPLICIT NONE
 USE distance_module
 TYPE(point)::p1,p2
 REAL::distance
 ....
 distance = p1 .dist. p2
 ....
END PROGRAM main
```


Operator definition: Example

```
MODULE distance_module
 IMPLICIT NONE
 TYPE point
  REAL::x,y
 END TYPPE point
 ....
 INTEFACE OPERATOR(.dist.)
  MODULE PROCEDURE calcdist
 END INTERFACE
 ....
CONTAINS

 REAL FUNCTION calcdist(px,py)
  TYPE(point), INTENT(IN)::px, py
  calcdist = SQRT((px%x-py%x)**2+ (px%y-py%y)**2)
 END FUNCTION calcdist
 ....
END MODULE distance_module
```


Assignment Overloading

- The assignement operator (=) could be overloaded in order to be used with derived type

```
REAL::ax
TYPE(point)::px
...
ax = px
...
```

- The subroutine should have two non optional arguments, the first argument should have INTENT(OUT) o INTENT(INOUT) attribute, while the second should have the INTENT(IN)

```
INTERFACE ASSIGNMENT(=)
  subroutine interface body
END INTERFACE
```


Assignment Overloading: Example

```
MODULE assignoverload_module
 IMPLICIT NONE
      ....
      TYPE point
        REAL::x,y
      END TYPPE point
      ....
      INTEFACE ASSIGNMENT(=)
        MODULE PROCEDURE assign_point
      END INTERFACE
      ....
CONTAINS
      SUBROUTINE assign_point(ax,px)
        REAL, INTENT(OUT)::ax
        TYPE(point), INTENT(IN)::px
        ax = MAX(px%x, px%y)
      END SUBROUTINE assign_point
 ....
END MODULE assignoverload_module
```

in the calling program:

```
....
USE assignoverload_mod
....
REAL::ax
TYPE(point)::px
....
Ax=px ! Type point to type
.... ! real now defined
```

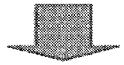

Exemple2

```
MODULE VECTOR_SPACE

 INTERFACE NORMA2
   MODULE PROCEDURE NORMA2_I, NORMA2_R
 END INTERFACE

CONTAINS

 INTEGER FUNCTION NORMA2_I(V)
   INTEGER, INTENT(IN) :: V(:)
   INTEGER I
   NORMA2_I = 0
   DO I=1,SIZE(V)
     NORMA2_I = NORMA2_I + V(I)**2
   END DO
 END FUNCTION NORMA2_I

 REAL(KIND=4) FUNCTION NORMA2_R(V)
   REAL(KIND=4), INTENT(IN) :: V(:)
   INTEGER I
   NORMA2_R = 0.0D0
   DO I=1,SIZE(V)
     NORMA2_R = NORMA2_R + V(I)**2
   END DO
 END FUNCTION NORMA2_R

END MODULE VECTOR_SPACE
```

Example2 (program)

```
PROGRAM EXEMPLE2
  USE VECTOR_SPACE
  INTEGER :: IV(3) = (/ 1, 2, 3 /)
  REAL(KIND=4)  :: RV(3) = (/ 1.0D0, 2.0D0, 3.0D0 /)
  INTEGER IN
  REAL(KIND=4)  RN

  IN = NORMA2(IV)
  RN = NORMA2(RV)

  WRITE(6,10) IN
  WRITE(6,11) RN
10  FORMAT('NORM of the INTEGER vector = ',I8)
11  FORMAT('NORM of the REAL    vector = ',F8.3)

  END PROGRAM EXEMPLE2
```


Overloading + Module



Data abstraction

Exemple 3

```
module data_types
 type vect
  real x,y,z
 end type vect
end module data_types

module data_functions
  Interface operator( + )
    module procedure my_sum
  end interface
 contains
  function my_sum (vect1,vect2) result ( new )
    use data_types
    type (vect), intent(IN) :: vect1
    type (vect), intent(IN) :: vect2
    type (vect), new
    new%x = vect1%x + vect2%x
    new%y = vect1%y + vect2%y
    new%z = vect1%z + vect2%z
  end function my_sum
end module data_functions
```


EXEMPLE 3 (program)

```
module algebra
  use data_types
  use data_functions
end module algebra

program exemple3
  use algebra
  implicit NONE
  type (vect) vect1,vect2,vect3
  vect1 = vect(1.0d0,2.0d0,3.0d0)
  vect2 = vect(3.0d0,2.0d0,1.0d0)
  vect3 = vect1 + vect2
  print *,'main :',vect3%x, vect3%y, vect3%z
end program
```


Goals:

- data hiding
- smooth migration
- performance
- modular code

FORTRAN 90

- modular programming paradigm
- includes F77 as a subset
- optimized compilers

Limitations

no object oriented

FORTRAN77

procedural programming paradigm

- focus on procedures
- no explicit dependencies among data
- no safe data grouping
- no data hiding

Alternative Languages

C++ , F90

F90 Modules

explicit dependencies (modules hierarchy)
contain data and methods (locality)
data hiding (safety)
interfaces (function prototyping and overloading)

Types + Modules

data abstraction
(better code design and readable codes)

Phase 1

- data structures analysis
- dependencies
- modules hierarchy

dependencies among data are explicit and coded into the data themself

Phase 2

- transform subroutines in sets of methods acting on data
- place methods in the modules of the data acted on

changes to the code are local
hierarchy guides you tracking the changes

Phase 3

- restrict the scope of the data
  (making then private to the modules)
- create interfaces
  (to access data within the modules)

data hiding
data abstraction

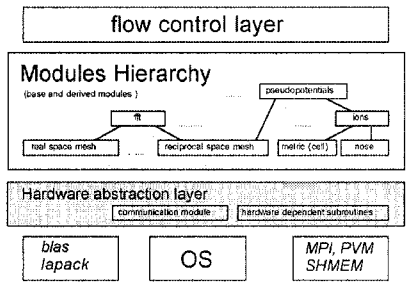
Advantages

easier to maintain, develop debug

suitable for a team of developers

locality increased

Code Structure



Make Command

If a code is large and/or it shares subroutines with other codes, it is useful to split the source in many files that could be placed in different directories.

In F90 there are dependencies among program units, i.e. modules must be compiled before than any other program units. Therefore there is a well defined order for compiling source files

To avoid compiling by hands the sources in the proper order, the ***make*** command could be used

Make Command

The make command can be programmed to do the job for you using a file containing instruction and dircetive.

By default the make command looks in the present directory for a file colled Makefile or makefile

A simple makefile

```
# this is a comment within the makefile

myprog.x : modules.o main.o
        f90 -o myprog.x modules.o main.o

modules.o : modules.f90
        f90 -c modules.f90

main.o : modules.o main.f90
        f90 -c main.f90
```

this tell to the make command that myprog.x depend from modules.o and main.o

make execute the command only when modules.o and main.o have been built

to compile the code, from the console the programmer issue the command:
> make

A less simple makefile

```
# this is a comment within the makefile

myprog.x : modules.o main.o
        f90 -o myprog.x modules.o main.o

main.o : modules.o

.f90.o
        f90 -c $<
```

this is an implicit dependency, it state that all files ".o" depend and should be generated from the corresponding ".f90" files

this is a make macro, and it is expandend with the proper ".f90" filename

In the above example, make try to built myprog.x but it realizes that main.o and modules.o should be generated first. Then it starts looking for a rule to make the ".o", and it finds that main.o depend on modules.o, and them make build an internal hierarchy for compilation in which modules.o come before main.o . Finally make finds the implicit rule and starts compiling the sources.