Advanced Concepts in Fortran 90

Alexander B. Pacheco

User Services Consultant
LSU HPC & LONI
sys-help@loni.org

LONI Workshop: Fortran Programming
Louisiana State University
Baton Rouge
Feb 13-16, 2012

Outline

Outline

Review I

Logical Structure

1. program name
2. declaration of variable types
3. read input
4. do calculations
5. write output
6. end program

Example Code

```
program hello
   implicit none
   character(len=100) :: your_name

   print *, 'Your name please'
   read *, your_name
   print *, 'Hello ', your_name

end program hello
```

Output

```
%>ifort -o hello hello.f90
%>./hello
  Your Name Please
  "Alex Pacheco"
  Hello Alex Pacheco
%>
```

Review II

Fortran Free Source Form

- Fortran 90/95/2003: free form source, a line can contain up to 132 characters
- Inline comments initiated by !
- Statements are continued by appending &
- program name and variables: up to 31 letters, digits and underscores (_)
- names must begin with a letter; digits and underscores are not allowed
- multiple commands on single line separated by semi-colon (;)

Review III

Coding Style

- always use **`implicit none`**
- avoid using mixed cases i.e. upper and lower case

  Some coders prefer fortran keywords, intrinsic functions and user defined entities as upper case while rest of the code in lower case. I prefer everything in lower case!

  For visibility, all intrinsic functions and user defined entities are in bold except when displaying a code available from the exercise directories
- Remember someone else will continue the development of your code, so

  INDENT your code, it makes it easier to read

  Add meaningfull comments where ever possible

Review IV

Declarations and Attributes

- Can state **implicit none**: all variables must be declared
- Syntax:

  `<type> [,<attribute-list>] [::] <variable-list> [=<value>]`

  `<type>` : data types i.e. integer, real, complex, character or logical

  `attributes` : dimension, parameter, pointer, target, allocatable, optional, intent

Examples of valid declarations

```
subroutine aroutine(x,i,j)
   implicit none
   real, intent(in) :: x
   logical :: what
   real,dimension(10,10) :: y, z(10)
   character(len=*),parameter :: somename
   integer, intent(out) :: i,j
   ...
end subroutine aroutine
```

Review V

Data Types

INTEGER:	exact whole numbers
REAL:	real, fractional numbers
COMPLEX:	complex, fractional numbers
LOGICAL:	boolean values
CHARACTER:	strings

Arithmetic Operators

+	: addition
-	: subtraction
*	: multiplication
/	: division
**	: exponentiation

Relational Operators

==	: equal to
/=	: not equal to
<	: less than
<=	: less than or equal to
>	: greater than
>=	: greater than or equal to

Logical Expressions

.TRUE.
.FALSE.
.AND.
.OR.
.NOT.

Operator Precedence

Operator	Precedence	Example
expression in ()	Highest	(a+b)
user-defined monadic	-	.inverse.a
**	-	10**4
* or /	-	10*20
monadic + or -	-	-5
dyadic + or -	-	1+5
//	-	str1//str2
relational operators	-	a > b
.not.	-	.not.allocated(a)
.and.	-	a.and.b
.or.	-	a.or.b
.eqv. or .neqv.	-	a.eqv.b
user defined dyadic	Lowest	x.dot.y

- x = a + b/5.0 - c**2 + 2.0*e
  exponentiation (**) has highest precedence followed by / and *
- The above expression is equivalent to
- x = a + b/5.0 - c’ + 2.0*e = a + b’ - c’ + 2.0*e = a + b’ - c’ + e’
  where b’ = b/5.0 , c’ = c**2 and e’ = 2.0*e
- x = a + b/5.0 - c**2 + (2.0*e)
- equivalent to x = a + b/5.0 - c**2 + e’ = a + b/5.0 - c’ + e’ = a + b’ - c’ + e’

Guidelines for Slides

Code in this block

Generic Code explaining Fortran Programming Structure or Style

Code in this block

Code in Exercises directory /work/apacheco/F90-workshop/Exercises

Code in this block

Code written only to explain content on current or previous slide

Code in this block

Code from Exercises directory but modified to describe content on current or previous slide

Output from code

Outline

Intrinsic Functions

- Fortran provides a set of intrinsic functions

Arithmetic Functions

Function	Action	Example
INT	conversion to integer	J=INT(X)
REAL	conversion to real	X=REAL(J)
CMPLX	conversion to complex	A=CMPLX(X,Y)
ABS	absolute value	Y=ABS(X)
MOD	remainder when I divided by J	K=MOD(I,J)
SQRT	square root	Y=SQRT(X)
EXP	exponentiation	Y=EXP(X)
LOG	natural logarithm	Y=LOG(X)
LOG10	logarithm to base 10	Y=LOG10(X)

Trignometric Functions

Function	Action	Example
SIN	sine	X=SIN(Y)
COS	cosine	X=COS(Y)
TAN	tangent	X=TAN(Y)
ASIN	arcsine	X=ASIN(Y)
ACOS	arccosine	X=ACOS(Y)
ATAN	arctangent	X=ATAN(Y)
ATAN2	arctangent(a/b)	X=ATAN2(A,B)

KIND Parameter I

- **kind** parameters provide a way to parameterize the selection of different possible machine representations for each intrinsic data types.
- The **kind** parameter is an integer which is processor dependent.
- There are only 2(3) kinds of reals: 4-byte, 8-byte (and 16-byte), respectively known as single, double (and quadruple) precision.
- The corresponding **kind** numbers are 4, 8 and 16 (most compilers)
- The value of the **kind** parameter is usually not the number of decimal digits of precision or range; on many systems, it is the number of bytes used to represent the value.
- The intrinsic functions **selected_int_kind** and **selected_real_kind** may be used to select an appropriate **kind** for a variable or named constant.

KIND Parameter II

```
program kind_function

  implicit none
  integer,parameter :: dp = selected_real_kind(15)
  integer,parameter :: ip = selected_int_kind(15)
  integer(kind=4) :: i
  integer(kind=8) :: j
  integer(ip) :: k
  real(kind=4) :: a
  real(kind=8) :: b
  real(dp) :: c

  print '(a,i2,a,i4)', 'Kind of i = ',kind(i), '  with range =', range(i)
  print '(a,i2,a,i4)', 'Kind of j = ',kind(j), '  with range =', range(j)
  print '(a,i2,a,i4)', 'Kind of k = ',kind(k), '  with range =', range(k)
  print '(a,i2,a,i2,a,i4)', 'Kind of real a = ',kind(a),&
       '  with precision = ', precision(a),&
       '  and range =', range(a)
  print '(a,i2,a,i2,a,i4)', 'Kind of real b = ',kind(b),&
       '  with precision = ', precision(b),&
       '  and range =', range(b)
  print '(a,i2,a,i2,a,i4)', 'Kind of real c = ',kind(c),&
       '  with precision = ', precision(c),&
       '  and range =', range(c)

end program kind_function
```

```
[apacheco@qb4 examples] ./kindfns
Kind of i =  4  with range =   9
Kind of j =  8  with range =  18
Kind of k =  8  with range =  18
Kind of real a =  4  with precision =  6  and range =  37
Kind of real b =  8  with precision = 15  and range = 307
Kind of real c =  8  with precision = 15  and range = 307
```

Outline

Control Constructs

- A Fortran program is executed sequentially

```
program somename
   variable declarations
   statement 1
   statement 2
   . . .
end program somename
```

- Control Constructs change the sequential execution order of the program
  1. Conditionals: IF
  2. Loops: DO
  3. Switches: SELECT/CASE
  4. Branches: GOTO (obsolete in Fortran 95/2003, use CASE instead)

If Statement

The general form of the `if` statement

`if` (*logical expression*) *statement*

- When the **`if`** statement is executed, the logical expression is evaluated.
- If the result is true, the statement following the logical expression is executed; otherwise, it is not executed.
- The statement following the logical expression **cannot** be another **`if`** statement. Use the **`if-then-else`** construct instead.

```
if (value < 0 ) value = 0
```

If-then-else Construct I

- The **`if-then-else`** construct permits the selection of one of a number of blocks during execution of a program
- The **if-then** statement is executed by evaluating the logical expression.
- If it is true, the block of statements following it are executed. Execution of this block completes the execution of the entire **if** construct.
- If the logical expression is false, the next matching **else if**, **else** or **end if** statement following the block is executed.

```
if (logical expression) then
    block of statements
else if (logical expression) then
    block of statements
else if ···
    .
    .
    .
else
    block of statements
end if
```

If-then-else Construct II

- Examples:

Letter Grade

```
if (x < 50 ) then
   GRADE = 'F'
else if (x < 60 ) then
   GRADE = 'D'
else if (x < 70 ) then
   GRADE = 'C'
else if (x < 80 ) then
   GRADE = 'B'
else
   GRADE = 'A'
end if
```

Find minimum of a,b and c

```
if (a < b .and.  a < c) then
   result = a
else if (b < a .and.  b < c ) then
   result = b
else
   result = c
end if
```

If-then-else Construct III

- The **else if** and **else** statements and blocks may be omitted.
- If **else** is missing and none of the logical expressions are true, the **if-then-else** construct has no effect.
- The **end if** statement must not be omitted.
- The **if-then-else** construct can be nested and named.

no else if

```
[construct name:] if (logical expression) then
    block of statements
else
    block of statements
    [name:] if (logical expression) then
        block of statements
    end if [name]
end if [construct name]
```

no else

```
if (logical expression) then
    block of statements
else if (logical expression) then
    block of statements
else if (logical expression) then
    block of statements
end if
```

Finding roots of quadratic equation I

```
program roots_of_quad_eqn

  implicit none

  real(kind=8) :: a,b,c
  real(kind=8) :: roots(2),d

  print *, '============================================'
  print *, ' Program to solve a quadratic equation'
  print *, '    ax^2 + bx + c = 0 '
  print *, ' If d = b^2 - 4ac >= 0 '
  print *, '   then solutions are: '
  print *, '      (-b +/- sqrt(d) )/2a '
  print *, '============================================'

  ! read in coefficients a, b, and c
  write(*,*) 'Enter coefficients a,b and c'
  read(*,*) a,b,c
  write(*,*)
  write(*,*) ' Quadratic equation to solve is: '
  write(*,fmt='(a,f5.3,a,f5.3,a,f5.3,a)') '   ',a,'x^2 + ',b,'x + ',c,' = 0'
  write(*,*)

  outer: if ( a == 0d0 ) then
     middle: if ( b == 0.d0 ) then
        inner: if ( c == 0.d0 ) then
           write(*,*) 'Input equation is 0 = 0'
        else
           write(*,*) 'Equation is unsolvable'
           write(*,fmt='(a,f5.3,a)') ' ',c,' = 0'
        end if inner
     else
        write(*,*) 'Input equation is a Linear equation with '
        write(*,fmt='(a,f6.3)') ' Solution: ', -c/b
     end if middle
  else
     d = b*b - 4d0*a*c
     dis0: if ( d > 0d0 ) then
```

Finding roots of quadratic equation II

```
        d = sqrt(d)
        roots(1) = -( b + d)/(2d0*a) ; roots(2) = -( b - d)/(2d0*a)
        write(*,fmt='(a,2f12.6)') 'Solution: ', roots(1),roots(2)
     else if ( d == 0.d0 ) then
        write(*,fmt='(a,f12.6)') 'Both solutions are equal: ', -b/(2d0*a)
     else
        write(*,*) 'Solution is not real'
        d = sqrt(abs(d))
        roots(1) = d/(2d0*a)
        roots(2) = -d/(2d0*a)
        write(*,fmt='(a,ss,f6.3,sp,f6.3,a2,a,ss,f6.3,sp,f6.3,a2)') &
             ' (',-b/(2d0*a),sign(roots(1),roots(1)),'i)',' and (',-b/(2d0*a),sign(roots(2),roots(2)),'i)'
     end if dis0
  end if outer
end program roots_of_quad_eqn
```

```
[apacheco@qb4 examples] ./root.x
 ============================================
  Program to solve a quadratic equation
     ax^2 + bx + c = 0
  If d = b^2 - 4ac >= 0
    then solutions are:
      (-b +/- sqrt(d) )/2a
 ============================================
 Enter coefficients a,b and c
1 2 1

  Quadratic equation to solve is:
   1.000x^2 + 2.000x + 1.000 = 0

Both solutions are equal:    -1.000000
```

```
[apacheco@qb4 examples] ./root.x
 ============================================
  Program to solve a quadratic equation
     ax^2 + bx + c = 0
  If d = b^2 - 4ac >= 0
    then solutions are:
      (-b +/- sqrt(d) )/2a
 ============================================
 Enter coefficients a,b and c
0 1 2

  Quadratic equation to solve is:
   0.000x^2 + 1.000x + 2.000 = 0

 Input equation is a Linear equation with
 Solution: -2.000
```

```
[apacheco@qb4 examples] ./root.x
 ============================================
  Program to solve a quadratic equation
     ax^2 + bx + c = 0
  If d = b^2 - 4ac >= 0
    then solutions are:
      (-b +/- sqrt(d) )/2a
 ============================================
 Enter coefficients a,b and c
2 1 1

  Quadratic equation to solve is:
    2.000x^2 +  1.000x +  1.000 = 0

 Solution is not real
 (-0.250+0.661i) and (-0.250-0.661i)
```

Case Construct I

- The **case** construct permits selection of one of a number of different block of instructions.
- The value of the expression in the **select case** should be an integer or a character string.

```
[construct name:] select case (expression)
   case (case selector)
     block of statements
   case (case selector)
     block of statements
     .
     .
     .
   [ case default
     block of statements ]
end select [construct name]
```

- The case selector in each **case** statement is a list of items, where each item is either a single constant or a range of the same type as the expression in the **select case** statement.
- A range is two constants separated by a colon and stands for all the values between and including the two values.
- The case default statement and its block are optional.

Case Construct II

- The **select case** statement is executed as follows:
    1. Compare the value of expression with the case selector in each case. If a match is found, execute the following block of statements.
    2. If no match is found and a **case default** exists, then execute those block of statements.

Notes

- The values in case selector must be unique.
- Use **case default** when possible, since it ensures that there is something to do in case of error or if no match is found.
- **case default** can be anywhere in the **select case** construct. The preferred location is the last location in the **case** list.

Case Construct III

case selector: character

```
select case (traffic_light)
  case ("red")
    print *, "Stop"
  case ("yellow")
    print *, "Caution"
  case ("green")
    print *, "Go"
  case default
    print *, "Illegal value:",&
      traffic_light
end select
```

case selector: integer

```
select case (month)
  case (1,3,5,7:8,10,12)
    number_of_days = 31
  case (4,6,9,11)
    number_of_days = 30
  case (2)
    if (leap_year) then
      number_of_days = 29
    else
      number_of_days = 28
    end if
end select
```

MD Code: Choose between Lennard-Jones or Morse Potential

```
select case(pot)
case("lj", "LJ")
   call ljpot(r,f,V)
case("mp", "MP")
   call morse(r,f,V)
case default
   call ljpot(r,f,V)
end select
```

Do Construct

- The looping construct in fortran is the **do construct**.
- The block of statements called the **loop body** or **do construct body** is executed repeatedly as indicated by loop control.
- A **do** construct may have a **construct name** on its first statement

Do Loop

[construct name:] `do` [*loop control*]
    block of statements
`end do` [construct name]

- There are two types of loop control:
  1. Counting: a variable takes on a progression of integer values until some limit is reached.
     - *variable = start, end[, stride]*
     - *stride* may be positive or negative integer, default is 1 which can be omitted.
  2. General: a loop control is missing

Do Construct: Counting I

- Before a **do** loop starts, the expression *start, end* and *stride* are evaluated. These values are not re-evaluated during the execution of the **do** loop.
- *stride* cannot be zero.
- If *stride* is positive, this **do** counts up.
  1. The *variable* is set to *start*
  2. If *variable* is less than or equal to *end*, the block of statements is executed.
  3. Then, *stride* is added to *variable* and the new *variable* is compared to *end*
  4. If the value of *variable* is greater than *end*, the **do** loop completes, else repeat steps 2 and 3
- If *stride* is negative, this **do** counts down.
  1. The *variable* is set to *start*
  2. If *variable* is greater than or equal to *end*, the block of statements is executed.
  3. Then, *stride* is added to *variable* and the new *variable* is compared to *end*
  4. If the value of *variable* is less than *end*, the **do** loop completes, else repeat steps 2 and 3

Do Construct: Counting II

```
program factorial2

  implicit none
  integer, parameter :: &
       dp = selected_int_kind(15)
  integer(dp) :: i,n,start,factorial

  print *, 'Enter an integer < 15 '
  read *, n

  if ( (n/2)*2 == n ) then
     start = 2 ! n is even
  else
     start = 1 ! n is odd
  endif
  factorial = 1_dp
  do i = start,n,2
     factorial = factorial * i
  end do
  write(*,'(i4,a,i15)') n,'!!=',factorial

end program factorial2
```

```
program factorial1

  implicit none
  integer, parameter :: dp = selected_int_kind(15)
  integer(dp) :: i,n,factorial

  print *, 'Enter an integer < 15 '
  read *, n

  factorial = n
  do i = n-1,1,-1
     factorial = factorial * i
  end do
  write(*,'(i4,a,i15)') n,'!=',factorial

end program factorial1
```

```
[apacheco@qb4 examples] ./fact2
 Enter an integer < 15
10
  10!!=            3840
```

```
[apacheco@qb4 examples] ./fact1
 Enter an integer < 15
10
  10!=        3628800
```

Do Construct: Nested

- The **exit** statement causes termination of execution of a loop.
- If the keyword **exit** is followed by the name of a do construct, that named loop (and all active loops nested within it) is exited.
- The **cycle** statement causes termination of the execution of *one iteration* of a loop.
- The **do** body is terminated, the **do** variable (if present) is updated, and control is transferred back to the beginning of the block of statements that comprise the **do** body.
- If the keyword **cycle** is followed by the name of a construct, all active loops nested within that named loop are exited and control is transferred back to the beginning of the block of statements that comprise the named **do** construct.

```
program nested_doloop

  implicit none
  integer,parameter :: dp = selected_real_kind(15)
  integer :: i,j
  real(dp) :: x,y,z,pi

  pi = 4d0*atan(1.d0)

  outer: do i =1,180
     inner: do j = 1,180
     x = real(i)*pi/180d0
     y = real(j)*pi/180d0
     if ( j == 90 ) cycle inner
     z = sin(x) / cos(y)
     print '(2i6,3f12.6)', i,j,x,y,z
     end do inner
  end do outer

end program nested_doloop
```

```
[apacheco@qb4 examples] ./nested
     0     0    0.000000    0.000000    0.000000
     0    45    0.000000    0.785398    0.000000
     0   135    0.000000    2.356194   -0.000000
     0   180    0.000000    3.141593   -0.000000
    45     0    0.785398    0.000000    0.707107
    45    45    0.785398    0.785398    1.000000
    45   135    0.785398    2.356194   -1.000000
    45   180    0.785398    3.141593   -0.707107
    90     0    1.570796    0.000000    1.000000
    90    45    1.570796    0.785398    1.414214
    90   135    1.570796    2.356194   -1.414214
    90   180    1.570796    3.141593   -1.000000
   135     0    2.356194    0.000000    0.707107
   135    45    2.356194    0.785398    1.000000
   135   135    2.356194    2.356194   -1.000000
   135   180    2.356194    3.141593   -0.707107
   180     0    3.141593    0.000000    0.000000
   180    45    3.141593    0.785398    0.000000
   180   135    3.141593    2.356194   -0.000000
   180   180    3.141593    3.141593   -0.000000
```

Do Construct: General

- The General form of a **do** construct is

```
[construct name:] do
      block of statements
end do [construct name]
```

- The *block of statements* will be executed repeatedly.
- To exit the **do** loop, use the **exit** or **cycle** statement.
- The **exit** statement causes termination of execution of a loop.
- The **cycle** statement causes termination of the execution of *one iteration* of a loop.

```
finite: do
   i = i + 1
   inner: if ( i < 10 ) then
      print *, i
      cycle finite
   end if inner
   if ( i > 100 ) exit finite
end do finite
```

Do While Construct

- If a condition is to be tested at the top of a loop, a **do ... while** loop can be used

do while (*logical expression*)
    block of statements
end do

- The loop only executes if the logical expression evaluates to `.true.`

```
finite: do while ( i <= 100 )
   i = i + 1
   inner: if ( i < 10 ) then
      print *, i
   end if inner
end do finite
```

```
finite: do
   i = i + 1
   inner: if ( i < 10 ) then
      print *, i
      cycle finite
   end if inner
   if ( i > 100 ) exit finite
end do finite
```

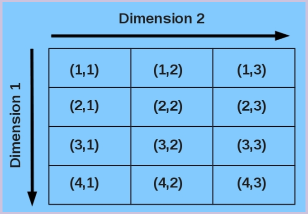
Outline

Arrays I

- Arrays (or matrices) hold a collection of different values at the same time.
- Individual elements are accessed by subscripting the array.
- A 10 element array is visualized as

1	2	3	⋯	8	9	10

while a 4x3 array as



- Each array has a type and each element of holds a value of that type.

Array Declarations

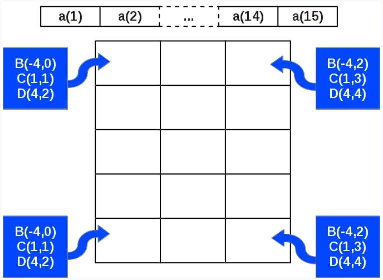
- The **dimension** attribute declares arrays.
- Usage: **dimension(lower_bound:upper_bound)**

  Lower bounds of one **(1:)** can be omitted
- Examples:
  ```
  integer, dimension(1:106) :: atomic_number
  real, dimension(3,0:5,-10:10) :: values
  character(len=3),dimension(12) :: months
  ```
- Alternative form for array declaration
  ```
  integer :: days_per_week(7), months_per_year(12)
  real :: grid(0:100,-100:0,-50:50)
  complex :: psi(100,100)
  ```
- Another alternative form which can be very confusing for readers
  ```
  integer, dimension(7) :: days_per_week, months_per_year(12)
  ```

Array Visualization

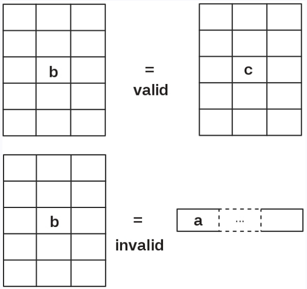
- Define arrays a, b, c and d as follows

```
real,dimension(15) :: a
real,dimension(-4:0,0:2) :: b
real,dimension(5,3) :: c
real,dimension(4:8,2:4) :: d
```



Array Conformance

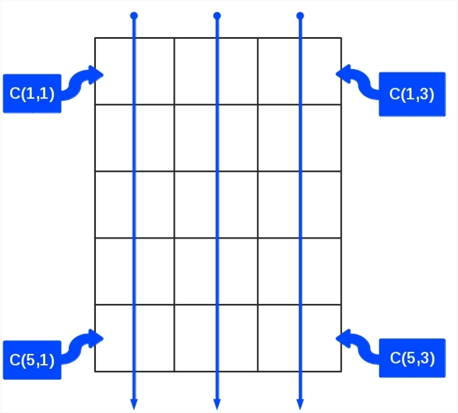
- Array or sub-arrays must conform with all other objects in an expression
  1. a scalar conforms to an array of any shape with the same value for every element
     `c = 1.0` is the same as `c(:,:) = 1.0`
  2. two array references must conform in their shape.



Arrays V

Array Element Ordering

- Fortran is a column major form i.e. elements are added to the columns seqeuntially. This ordering can be changed using the **`reshape`** intrinsic.



Array Terminology

```
real :: a(0:20), b(3,0:5,-10:10)
```

Rank:	Number of dimensions.
	`a` has rank 1 and `b` has rank 3
Bounds:	upper and lower limits of each dimension of the array.
	`a` has bounds 0:20 and `b` has bounds 1:3, 0:5 and -10:10
Extent:	Number of element in each dimension
	`a` has extent 21 and `b` has extents 3,6 and 21
Size:	Total number of elements.
	`a` has size 21 and `b` has 30
Shape:	The shape of an array is its rank and extent
	`a` has shape 21 and `b` has shape (3,6,21)

- Arrays are conformable if they share a shape.
- The bounds do not have to be the same

```
c(4:6) = d(1:3)
```

Array Constructors

- Used to give arrays or sections of arrays specific values

```
implicit none
integer :: i
integer, dimension(10) :: imatrix
character(len=5),dimension(3) :: colors
real, dimension(4) :: height
height = (/5.10, 5.4, 6.3, 4.5 /)
colors = (/'red ', 'green', 'blue ' /)
ints = (/ 30, (i = 1, 8), 40 /)
```

- constructors and array sections must conform.

  `ints = (/ 30, (i = 1, 10), 40/)` is invalid
- strings should be padded so that character variables have correct length.
- use **`reshape`** intrinsic for arrays for higher ranks
- `(i = 1, 8)` is an implied **do**.
- You can also specify a stride in the implied **do**.

  `ints = (/ 30, (i = 1, 16, 2), 40 /)`
- There should be no space between / and (or)

Reshape

- **reshape(source, shape, pad, order)** constructs an array with a specified shape **shape** starting from the elements in a given array **source**.
- If **pad** is not included then the size of **source** has to be at least **product (shape)**.
- If **pad** is included it has to have the same type as **source**.
- If **order** is included, it has to be an **integer** array with the same shape as **shape** and the values must be a permutation of (1,2,3,...,N), where N is the number of elements in **shape** , it has to be less than, or equal to 7.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & a & a \\ a & 0 & a \\ a & a & 0 \end{pmatrix}$$

```
rcell = reshape( (/ &
        0.d0, 0.d0, a,    a,    &
        0.d0, a,    0.d0, a,    &
        0.d0, a,    a,    0.d0 &
        /),(/4,3/) )
```

```
rcell = reshape( (/ &
        0.d0, 0.d0, 0.d0 &
        0.d0, a    , a    &
        a,    0.d0, a    &
        a,    a,    0.d0 &
        /),(/4,3/),order=(/2,1/))
```

In Fortran, for a multidimensional array, the first dimension has the fastest index while the last dimension has the slowest index i.e. memory locations are continuous for the last dimension. The **order** statement allows the programmer to change this order. The last example above sets the memory location order which is consistent to that in C/C++.

Array Contructors in Intialization Statements

- Arrays can be initialized as follows during variable declaration

```
integer, dimension(4) :: imatrix = (/ 2, 4, 6, 8/)
character(len=*),dimension(3) :: colors = (/'red ', 'green', 'blue '/)
```

All strings must be the same length

```
real, dimension(4) :: height = (/5.10, 5.4, 6.3, 4.5/)
integer, dimension(10) :: ints = (/ 30, (i = 1, 8), 40/)
real, dimension(4,3), parameter :: rcell =reshape( (/0.d0, 0.d0, 0.d0, 0.d0,&
   a, a, a,0.d0, a, a, a, 0.d0 /),(/4,3/),order=(/2,1/))
```

Array Syntax I

- whole arrays
  - `a = 0.0`
    sets whole array `a` to zero
  - `b = c + d`
    adds `c` to `d` and assigns result to `b`
- elements
  - `a(1) = 0.0`
    sets one element of `a` to zero
  - `b(1,3) = a(3) + c(5,1)`
    sets an element of `b` to the sum of two other array elements.
- array sections
  - `a(0:3) = 5.0`
    sets `a(0)`, `a(1)`, `a(2)` and `a(3)` to five
  - `b(-2:2,4:6) = c(1:5,6:8) + 2.0`
    adds two to the subsection of `c` and assigns the result to the subsection of `b`

Array Syntax II

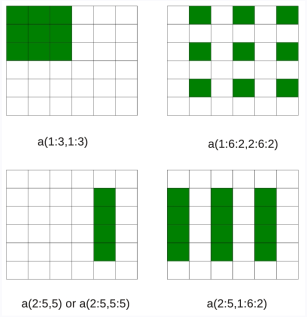
- Arrays can be treated as a single variable:
  - can use intrinsic operators between conformable arrays (or sections)
    ```
     b = c * d + b**2
    ```
    this is equivalent to
    ```
     b(-4,0) = c(1,1) * d(4,2) + b(-4,0)**2
     b(-3,0) = c(2,1) * d(5,2) + b(-3,0)**2
    ...
     b(-4,0) = c(1,1) * d(4,2) + b(-4,0)**2
     b(-4,1) = c(1,2) * d(4,3) + b(-4,1)**2
    ...
     b(-3,2) = c(4,3) * d(7,4) + b(-3,2)**2
     b(-4,2) = c(5,3) * d(8,4) + b(-4,2)**2
    ```
  - elemental intrinsic functions can be used
    ```
     b = sin(c) + cos(d)
    ```

    All operations/functions are applied element by element

Array Sections I

```
real, dimension(6:6) :: a
```

- `a(1:3,1:3) = a(1:6:2,2:6:2)` and `a(1:3,1:3) = 1.0` are valid
- `a(2:5,5) = a(2:5,1:6:2)` and `a(2:5,1:6:2) = a(1:6:2,2:6:2)` are not
- `a(2:5,5)` is a 1D section while `a(2:5,1:6:2)` is a 2D section



- The general form for specifying sub-arrays or sections is *[<bound1>]:[<bound2>][:<stride>]*
- The section starts at <*bound1*> and ends at or before <*bound2*>.
- <*stride*> is the increment by which the locations are selected, by default *stride=1*
- <*bound1*>, <*bound2*>, <*stride*> must all be scalar integer expressions.

Array Sections II

```
real, dimension(1:20) :: a
integer :: m,n,k
```

`a(:)`	the whole array
`a(3:9)`	elements 3 to 9 in increments of 1
`a(3:9:1)`	as above
`a(m:n)`	elements m through n
`a(m:n:k)`	elements m through n in increments of k
`a(15:3:-2)`	elements 15 through 3 in increments of -2
`a(15:3)`	zero size array
`a(m:)`	elements m through 20, default upper bound
`a(:n)`	elements 1, default lower bound through n
`a(::2)`	all elements from lower to upper bound in increments of 2
`a(m:m)`	1 element section
`a(m)`	array element not a section

are valid sections.

Array I/O **I**

real,dimension(4,4) :: a

- Arrays are printed in the order that they appear in memory
- ♦ `print *, a`

  would produce on output

  `a(1,1),a(2,1),a(3,1),a(4,1),a(1,2),a(2,2),···,a(3,4),a(4,4)`
- ♦ `read *, a`

  would read from input and assign array elements in the same order as above
- The order of array I/O can be changed using intrinsic functions such as **reshape**, **transpose** or **cshift**.

Array I/O II

Example

- Consider a 3x3 matrix

1	4	7
2	5	8
3	6	9

- The following print statements

```
print *, 'array element = ',a(3,3)
print *, 'array section = ',a(:,2)
print *, 'sub-array = ',a(:3,:2)
print *, 'whole array = ',a
print *, 'array transpose = ',transpose(a)
```

- would produce the following output

```
array element = 9
array section = 4 5 6
sub-array = 1 2 3 4 5 6
whole array = 1 2 3 4 5 6 7 8 9
array transpose = 1 4 7 2 5 8 3 6 9
```

Array Intrinsic Functions I

size(x[,n])	The size of x (along the n^{th} dimension, optional)
shape(x)	The shape of x
lbound(x[,n])	The lower bound of x
ubound(x[,n])	The upper bound of x
minval(x)	The minimum of all values of x
maxval(x)	The maximum of all values of x
minloc(x)	The indices of the minimum value of x
maxloc(x)	The indices of the maximum value of x

Array Intrinsic Functions II

sum(x[,n])	The sum of all elements of x (along the n^{th} dimension, optional) $sum(x) = \sum_{i,j,k,\cdots} x_{i,j,k,\cdots}$
product(x[,n])	The product of all elements of x (along the n^{th} dimension, optional) $prod(x) = \prod_{i,j,k,\cdots} x_{i,j,k,\cdots}$
transpose(x)	Transpose of array x: $x_{i,j} \Rightarrow x_{j,i}$
dot_product(x,y)	Dot Product of arrays x and y: $\sum_i x_i * y_i$
matmul(x,y)	Matrix Multiplication of arrays x and y which can be 1 or 2 dimensional arrays: $z_{i,j} = \sum_k x_{i,k} * y_{k,j}$
conjg(x)	Returns the conjugate of x: $a + \imath b \Rightarrow a - \imath b$

Allocatable Arrays I

Why?

- At compile time we may not know the size an array needs to be
- We may want to change the problem size without recompiling
- The molecular dynamics code was written for 108 atoms. If you want to run a simulation for 256 and 1024 atoms, do you need to recompile and create two executables?

Allocatable Arrays II

- Allocatable arrays allow us to set the size at run time.

```
real, allocatable :: force(:,:)
real, dimension(:),allocatable :: vel
```

- We set the size of the array using the allocate statement.

```
allocate(force(natoms,3))
```

- We may want to change the lower bound for an array

```
allocate(grid(-100,100))
```

- We may want to use an array once somewhere in the program, say during initialization. Using allocatable arrays also us to dynamically create the array when needed and when not in use, free up memory using the **`deallocate`** statement

```
deallocate(force,grid)
```

Allocatable Arrays III

- Sometimes, we want to check whether an array is allocated or not at a particular part of the code
- Fortran provides an intrinsic function, **`allocated`** which returns a scalar logical value reporting the status of an array

```
if ( allocated(grid))deallocate(grid)
if ( .not.  allocated(force)) allocate(force(natoms,3))
```

Masked Array Assignment: Where Statement

- Masked array assignment is achieved using the **where** statement

  where (c < 2) a = b/c

  the left hand side of the assignment must be array valued.

  the mask, (logical expression) and the right hand side of the assignment must all conform
- Fortran 95/2003 introduced the **where ... elsewhere ... end where** functionality
- **where** statement cannot be nested

MD code: subroutine integrate

```
where ( r > boxl )
r = r - boxl
end where
where ( r < 0d0 )
   r = r + boxl
end where
```

original code: subroutine integrate

```
   do j=1,3
      if (r(j) .gt. boxl(j)) then
         r(j) = r(j) - boxl(j)
      endif

      if (r(j) .lt. 0.d0) then
         r(j) = r(j) + boxl(j)
      endif
   enddo
enddo
```

Vector valued subscripts

- A 1D array can be used to subscript an array in a dimension

```
real, dimension(15) :: a
integer, dimension(5) :: v = (/ 1,4,8,10,15/)
integer, dimension(3) :: w = (/ 1,2,3/)
```

- ◆ `a(v)` is `a(1), a(4), a(8), a(10)` and `a(15)`
- ◆ `a(v) = 1.2` is valid
- ◆ only 1D vector subscripts are allowed

```
a(1) = prod(c(v,w))
```

Break

Outline

Program Units I

- Most programs are hundreds or more lines of code.
- Use similar code in several places.
- A single large program is extremely difficult to debug and maintain.
- Solution is to break up code blocks into procedures

Subroutines:	Some out-of-line code that is called exactly where it is coded
Functions:	Purpose is to return a result and is called only when the result is needed
Modules:	A module is a program unit that is not executed directly, but contains data specifications and procedures that may be utilized by other program units via the use statement.

Program Units II

```
program main

  use module1

  implicit none
  variable declarations


  ...
  call routine1(arg1,arg2,arg3)
  ...
  abc = func(arg1,arg2)
  ...

  contains

    subroutine routine1(arg1,arg2)
      ...
    end subroutine routine1

    function func(arg1,arg2)
      ...
    end function func

  end program main
```

```
program name

  specify which modules to use

  variable declarations


  block of statements

  call subroutine routine1 with arguments
  block of statements
  abc is some function of arg1 and arg2
  block of statements

  contains internal procedures described below

    contents of subroutine routine1
      ...


    contents of function func
      ...


  last end statement
```

MD Main Program

```
program md
  use param

  implicit none
  integer :: k, iseed(1)
  real(dp) :: pener,v2t,etot,avtemp
  real(dp),dimension(:,:),allocatable :: coord,coord0,vel,force

  interface
     subroutine setup(coord,vel,coord0)
       use param,only:dp
       implicit none
       real(dp),dimension(:,:),intent(out) :: coord,coord0,vel
     end subroutine setup
     subroutine verlet(coord,force,pener)
       use param,only:dp
       implicit none
       real(dp),dimension(:,:),intent(in) :: coord
       real(dp),dimension(:,:),intent(out) :: force
       real(dp),intent(out) :: pener
     end subroutine verlet
     subroutine integrate(coord,force,coord0,vel)
       use param,only:dp
       implicit none
       real(dp),dimension(:,:),intent(inout)::coord,coord0,vel
       real(dp),dimension(:,:),intent(in)::force
     end subroutine integrate
     subroutine rescale(vel)
       use param,only:dp
       implicit none
       real(dp),dimension(:,:),intent(inout) :: vel
     end subroutine rescale
  end interface
```

```
  inp=40
  outp=50

  iseed(1) = 12345
  call random_seed
  call random_seed(size=k)
  call random_seed(put=iseed(1:k))
  call init
  iprint = nstep / 10
  allocate(coord(npart,3),coord0(npart,3),vel(npart,3),force(npart,3))
  call setup(coord,vel,coord0)
  do istep = 1,nstep
     call verlet(coord,force,pener)
     call integrate(coord,force,coord0,vel)
     v2t = 0.d0
     !   Can you use intrinsic functions to simplify this calculation
     do i=1,npart
        v2t = v2t + dot_product(vel(i,:),vel(i,:))
     enddo
     etot = pener + 0.5d0*v2t
     avtemp = v2t / real(3 * npart, dp )
     ! output energies/velocities, deal with this later
     write(44,1000)real(istep,dp)*tstep,pener,v2t,etot,avtemp
     !         if (istep .gt. iprint) then
     !            if (mod(istep,500) .eq. 0) then
     call rescale(vel)
     !            endif
     !         endif
1000 format(5(1x,1pe15.8,1x))
  enddo
end program md
```

Subroutines I

CALL Statement

The **`call`** statement evaluates its arguments and transfers control to the subroutine
Upon return, the next statement is executed.

SUBROUTINE Statement

The **`subroutine`** statement declares the procedure and its arguments.
These are also known as dummy arguments.

The subroutine's interface is defined by

- The **`subroutine`** statement itself
- The declarations of its dummy arguments
- Anything else that the subroutine uses
- In the previous example, the **`subroutine verlet`** is an external procedure and can be called by any program unit with the program.

Subroutines II

Statement Order

1. A **`subroutine`** statement starts a subroutine
2. Any **`use`** statements come next
3. **`implicit none`** comes next, followed by
4. rest of the declarations,
5. executable statements
6. End with a **`end subroutine`** statement

Dummy Arguments

- Their names exist only in the procedure and are declared as local variables.
- The dummy arguments are associated with the actual arguments passed to the subroutines.
- The dummy and actual argument lists must match, i.e. the number of arguments must be the same and each argument must match in type and rank.

Subroutines III

Example

```
subroutine verlet(coord_t,force_t,pener)

  use param,only : dp,npart,boxl
  use lennjones

  implicit none
  integer :: i,j
  real(dp)  :: f(3),r(3)
  real(dp),dimension(:,:),intent(in) :: coord_t
  real(dp),dimension(:,:),intent(out) :: force_t
  real(dp),intent(out) :: pener

  pener = 0.d0
  force_t = 0d0

  do i=1,npart-1
     do j=i+1,npart
        r(:) = coord_t(i,:) - coord_t(j,:)
        !
        ! periodic boundary conditions
        !
        r(:) = r(:) - nint(r(:)/boxl(:))*boxl(:)
        !
        ! calculate lennard-jones forces and energies
        !
        r2 = 1.0d0 / (r(1)*r(1) + r(2)*r(2) + r(3)*r(3))
        r6 = r2 * r2 * r2

        f(:) = dvdr(r2,r6)*r(:)
        force_t(i,:) = force_t(i,:) + f
        force_t(j,:) = force_t(j,:) - f
        pener = pener + epot(r2,r6)

     enddo
  enddo
end subroutine verlet
```

How It's Called

```
program md

  use param
  implicit none
  integer :: k,iseed(1)
  real(dp) :: pener,v2t,etot,avtemp
  real(dp),dimension(:,:),allocatable :: coord,coord0,vel,force

  interface
     ...
     subroutine verlet(coord,force,pener)
       use param,only:dp
       implicit none
       real(dp),dimension(:,:),intent(in) :: coord
       real(dp),dimension(:,:),intent(out) :: force
       real(dp),intent(out) :: pener
     end subroutine verlet
     ...
  end interface

  inp=40
  outp=50

  iseed(1) = 12345
  call random_seed
  call random_seed(size=k)
  call random_seed(put=iseed(1:k))
  call init
  iprint = nstep / 10
  allocate(coord(npart,3),coord0(npart,3),vel(npart,3),force(npart,3))
  call setup(coord,vel,coord0)
  do istep = 1,nstep
     call verlet(coord,force,pener)
     call integrate(coord,force,coord0,vel)
     ...
  enddo
end program md
```

Internal Procedures

- Internal procedures appear just before the last **`end`** statement and are preceeded by the **`contains`** statement.
- Internal procedures can be either subroutines or functions which can be accessed only by the program, subroutine or module in which it is present
- Internal procedures have declaration of variables passed on from the parent program unit
- If an internal procedure declares a variable which has the same name as a variable from the parent program unit then this supersedes the variable from the outer scope for the length of the procedure.

Functions

- **`function`**s operate on the same principle as **`subroutine`**s
- The only difference is that **`function`** returns a value and does not require the **`call`** statement

Example

```
module lennjones
  use precision
  implicit none
  real(dp) :: d2,d6

  contains
    function dvdr(d2,d6)
      implicit none
      real(dp) :: dvdr
      dvdr = 48*d2*d6*(d6 - 0.5d0)
    end function dvdr
    function epot(d2,d6)
      implicit none
      real(dp) :: epot
      epot = 4.d0*d6*(d6 - 1.d0)
    end function epot
end module lennjones
```

How It's Called

```
subroutine verlet(coord,force,pener)
  use param,only : dp,npart,boxl
  use lennjones

  implicit none
  integer :: i,j
  real(dp) :: f(3),r(3)
  real(dp),dimension(:,:),intent(in) :: coord
  real(dp),dimension(:,:),intent(out) :: force
  real(dp),intent(out) :: pener

  pener = 0.d0
  force = 0d0
  do i=1,npart-1
     do j=i+1,npart
        r(:) = coord(i,:) - coord(j,:)
        !     periodic boundary conditions
        r(:) = r(:) - nint(r(:)/boxl(:))*boxl(:)
        !     calculate lennard-jones forces and energies
        r2 = 1.0d0 / (r(1)*r(1) + r(2)*r(2) + r(3)*r(3))
        r6 = r2 * r2 * r2

        f(:) = dvdr(r2,r6)*r(:)
        force(i,:) = force(i,:) + f
        force(j,:) = force(j,:) - f
        pener = pener + epot(r2,r6)
     enddo
  enddo
end subroutine verlet
```

Array-valued Functions

- `function` can also return arrays

Example

```
module lennjones
  use precision
  implicit none
  real(dp) :: d2,d6

  contains
    function dvdr(r,d2,d6)
      implicit none
      real(dp),dimension(:),intent(in) :: r
      real(dp),dimension(size(r,1)) :: dvdr
      real(dp) :: dvr
      dvr = 48*d2*d6*(d6 - 0.5d0)
      dvdr = dvr * r
    end function dvdr
    function epot(d2,d6)
      implicit none
      real(dp) :: epot
      epot = 4.d0*d6*(d6 - 1.d0)
    end function epot
end module lennjones
```

How It's Called

```
subroutine verlet(coord,force,pener)
  use param,only : dp,npart,boxl
  use lennjones

  implicit none
  integer :: i,j
  real(dp) :: f(3),r(3)
  real(dp),dimension(:,:),intent(in) :: coord
  real(dp),dimension(:,:),intent(out) :: force
  real(dp),intent(out) :: pener

  pener = 0.d0
  force = 0d0
  do i=1,npart-1
     do j=i+1,npart
        r(:) = coord(i,:) - coord(j,:)
        !     periodic boundary conditions
        r(:) = r(:) - nint(r(:)/boxl(:))*boxl(:)
        !     calculate lennard-jones forces and energies
        r2 = 1.0d0 / (r(1)*r(1) + r(2)*r(2) + r(3)*r(3))
        r6 = r2 * r2 * r2

        force(i,:) = force(i,:) + dvdr(r,r2,r6)
        force(j,:) = force(j,:) - dvdr(r,r2,r6)
        pener = pener + epot(r2,r6)
     enddo
  enddo
end subroutine verlet
```

Recursive Procedures

- In Fortran 90, recursion is supported as a feature
  1. **`recursive`** procedures call themselves
  2. **`recursive`** procedures must be declared explicitly
  3. **`recursive function`** declarations must contain a **`result`** keyword, and
  4. one type of declaration refers to both the function name and the result variable.

```
program fact

  implicit none
  integer :: i
  print *, 'enter integer whose factorial you want to calculate'
  read *, i

  print '(i5,a,i20)', i, '! = ', factorial(i)

contains
  recursive function factorial(i) result(i_fact)
    integer, intent(in) :: i
    integer :: i_fact

    if ( i > 0 ) then
       i_fact = i * factorial(i - 1)
    else
       i_fact = 1
    end if
  end function factorial

end program fact
```

```
[apacheco@qb4 examples] ./factorial
 enter integer whose factorial you want to calculate
10
   10! =              3628800
[apacheco@qb4 examples] ./fact1
 Enter an integer < 15
10
  10!=        3628800
```

Argument Association

- Recall from MD code example the invocation
  ```
  call subroutine verlet(coord,force,pener)
  ```
- and the subroutine declaration
  ```
  subroutine verlet(coord_t,force_t,pener)
  ```

- `coord` is an actual argument and is associated with the dummy argument `coord_t`
- In `subroutine verlet`, the name `coord_t` is an alias for `coord`
- If the value of a dummy argument changes, then so does the value of the actual argument
- Also, recall the `dvdr` function on the previous slide.
- The actual and dummy arguments must correspond in type, kind and rank.

Local Objects

- In **subroutine** verlet,

  i, j, r and f are local objects.
- Local Objects
  - are created each time a procedure is invoked
  - are destroyed when the procedure completes
  - do not retain their values between calls
  - do not exist in the programs memory between calls.

Example

```
subroutine verlet(coord,force,pener)

  use param,only : dp,npart,boxl
  use lennjones

  implicit none
  integer :: i,j
  real(dp) :: f(3),r(3)
  real(dp),dimension(:,:),intent(in) :: coord
  real(dp),dimension(:,:),intent(out) :: force
  real(dp),intent(out) :: pener

  ...

end subroutine verlet
```

Keyword Arguments

- Keyword Arguments
  - allow arguments to be specified in any order
  - makes it easy to add an extra argument - no need to modify any calls
  - helps improve readability of the program
  - are used when a procedure has optional arguments
- once a keyword is used, all subsequent arguments must be keyword arguments
- if used with external procedures then the **`interface`** must be explicit within the procedure in which it is invoked.

```
subroutine verlet(coord_t,force_t,pener_t)

  ...
  real(dp),intent(in),dimension(:,:) :: coord_t
  real(dp),intent(out),dimension(:,:) :: force_t
  real(dp),intent(out) :: pener_t
  ...
end subroutine force
```

```
program md
         ...
     call verlet(coord,force,pener)
         ...
end program md
```

- **`program`** `md` can invoke **`subroutine`** `verlet` using
  1. using the positional argument invocation (see right block)
  2. using keyword arguments

     `call` `force(force_t=force, pener_t=pener, coord_t=coord)`
     `call` `force(coord, force_t=force, pener_t=pener)`

Optional Arguments

- Optional Arguments
  - allow defaults to be used for missing arguments
  - make some procedures easier to use
- once an argument has been omitted all subsequent arguments must be keyword arguments
- the **`present`** intrinsic can be used to check for missing arguments
- if used with external procedures then the **`interface`** must be explicit within the procedure in which it is invoked.

Dummy Array Arguments

- There are two main types of dummy array argument:
  1. *explicit-shape*: all bounds specified
     - `real, dimension(4,4), intent(in) :: explicit_shape`
       The actual argument that becomes associated with an explicit shape dummy must conform in size and shape
  2. *assumed-shape*: no bounds specified, all inherited from the actual argument
     - `real, dimension(:,:), intent(out) :: assumed_shape`
     - An explicit interface must be provided
- dummy arguments cannot be (unallocated) allocatable arrays.

Explicit-shape Arrays

```
program md

  implicit none
  integer :: inp,outp,nstep,istep,iprint,i,j,nunit,npart
  real(kind=8) :: boxl(3),tstep,temp,avtemp
  real(kind=8) :: pener,v2t,etot
  real(kind=8),dimension(:,:),allocatable :: coord,coord0,vel,force

  ...
  allocate(coord(npart,3),coord0(npart,3),vel(npart,3),force(npart,3))
  call setup(coord,vel,npart,nunit,boxl,coord0,tstep,temp)
  ...
end program md
```

```
subroutine setup(coord,vel,npart,nunit,boxl,coord0,tstep,temp)

  implicit none
  real(kind=8) :: coord(npart,3)
  real(kind=8) :: vel(npart,3)
  real(kind=8) :: coord0(npart,3)
  ...
end subroutine setup
```

Assumed-Shape Arrays

```
program md

  use param

  implicit none
  real(dp) :: pener,v2t,etot,avtemp
  real(dp),dimension(:,:),allocatable :: coord,coord0,vel,force

  interface
     subroutine setup(coord,vel,coord0)
       use param,only:dp
       implicit none
       real(dp),dimension(:,:),intent(out) :: coord,coord0,vel
     end subroutine setup
     ...
  end interface

  ...
  allocate(coord(npart,3),coord0(npart,3),vel(npart,3),force(npart,3))
  call setup(coord,vel,coord0)
  ...
end program md
```

```
subroutine setup(coord,vel,coord0)

  use param, only : dp,npart,boxl,tstep,temp,nunit

  implicit none
  real(dp),dimension(:,:),intent(out) :: coord,coord0,vel
  ...
end subroutine setup
```

- The actual arguments cannot be vector subscribed array.
- The actual argument cannot be an assumed-size array
- In the procedure, bounds begin at 1
- If using external procedure, an explicit interface must be described

Automatic Arrays

- Automatic Arrays: Arrays which depend on dummy arguments
  1. their size is determined by dummy arguments
  2. they cannot have the **save** attribute or be initialized.
- The **size** intrinsic or dummy arguments can be used to declare automatic arrays.

```
program main
  implicit none
    integer :: i,j
    real, dimension(5,6) :: a
    ...
    call routine (a,i,j)
    ...
  contains
    subroutine routine(c,m,n)
      integer :: m,n
      real, dimension(:,:), intent(inout) :: c ! assumed shape array
      real :: b1(m,n) ! automatic array
      real, dimension(size(c,1),size(c,3)) :: b2 ! automatic array
      ...
    end subroutine routine
end program main
```

Save Attribute and Arrays

- Declaring a variable (or array) as **save** gives it a static storage memory.
- i.e information about variables is retained in memory between procedure calls.

```
subroutine something(iarg1)
  implicit none
    integer, intent(in) :: iarg1
    real,dimension(:,:),allocatable,save :: a
    real, dimension(:,:),allocatable :: b
    ...
    if (.not.allocated(a))allocate(a(i,j))
    allocate(b(j,i))
    ...
    deallocate(b)
end subroutine something
```

- Array `a` is saved when `something` exits.
- Array `b` is not saved and needs to be allocated every time in `something` and deallocated, to free up memory, before `something` exits.

Intent

- **`intent`** attribute was introduced in Fortran 90 and is recommended as it
  1. allows compilers to check for coding errors
  2. facilitates efficient compilation and optimization
- Declare if a parameter is
  - Input: **`intent(in)`**
  - Output: **`intent(out)`**
  - Both: **`intent(inout)`**

```
subroutine integrate(coord,coord0,force,vel)
  use precision
  implicit none
  real(dp),intent(inout),dimension(:,:) :: coord,coord0
  real(dp),intent(in),dimension(:,:) :: force
  real(dp),intent(out),dimension(:,:) :: vel
  ...
end subroutine integrate
```

- A variable declared as **`intent(in)`** in a procedure cannot be changed during the execution of the procedure (see point 1 above)

Interfaces I

- The **`interface`** statement is the first statement in an interface block.
- The **`interface`** block is a powerful structure that was introduced in FORTRAN 90.
- When used, it gives a calling procedure the full knowledge of the types and characteristics of the dummy arguments that are used inside of the procedure that it references.
- This can be a very good thing as it provides a way to execute some safety checks when compiling the program.
- Because the main program knows what argument types should be sent to the referenced procedure, it can check to see whether or not this is the case.
- If not, the compiler will return an error message when you attempt to compile the program.

Interfaces II

```
subroutine verlet(coord0,coord,vel,force,pener)
  use param,only : dp,npart,tstep,boxl

  implicit none
  integer :: i,j
  real(dp) :: r(3)
  real(dp),dimension(:,:),intent(inout)::coord,coord0
  real(dp),dimension(:,:),intent(out)::vel,force
  real(dp),intent(out)::pener

  interface
     subroutine pot_force(coord,force,pener)
       use precision
       implicit none
       real(dp),dimension(:,:),intent(in) :: coord
       real(dp),dimension(:,:),intent(out) :: force
       real(dp),intent(out) :: pener
     end subroutine pot_force
  end interface
  ! get potential and force
  call pot_force(coord,force,pener)
  !     update positions using the verlet algorithm
  do i=1,npart
     r(:) = 2*coord(i,:) - coord0(i,:) + force(i,:)*tstep*tstep
     vel(i,:) = (r(:) - coord0(i,:)) / (2*tstep)
     !     periodic boundary conditions
     where ( r > boxl )
        r = r - boxl
     end where
     where ( r < 0d0 )
        r = r + boxl
     end where
     ! update coordinates
     coord0(i,:) = coord(i,:)
     coord(i,:) = r(:)
  enddo
  !   output coordinates
  ...
end subroutine verlet
```

```
subroutine pot_force(coord,force,pener)
  use param,only : dp,npart,boxl,pot
  use lennjones

  implicit none
  integer :: i,j
  real(dp) :: f(3),r(3),V
  real(dp),dimension(:,:),intent(in) :: coord
  real(dp),dimension(:,:),intent(out) :: force
  real(dp),intent(out) :: pener

  pener = 0.d0
  force = 0d0

  do i=1,npart-1
     do j=i+1,npart
        r(:) = coord(i,:) - coord(j,:)
        !     periodic boundary conditions
        r(:) = r(:) - nint(r(:)/boxl(:))*boxl(:)
        !     calculate lennard-jones forces and energies
        select case(pot)
        case("lj", "LJ")
           call ljpot(r,f,V)
        case("mp", "MP")
           call morse(r,f,V)
        case default
           call ljpot(r,f,V)
        end select
        pener = pener + V
        force(i,:) = force(i,:) + f(:)
        force(j,:) = force(j,:) - f(:)
     enddo
  enddo

end subroutine pot_force
```

Interfaces III

```
subroutine verlet(coord0,coord,vel,force,pener)
  use param,only : dp,npart,tstep,boxl

  implicit none
  integer :: i,j
  real(dp) :: r(3)
  real(dp),dimension(:,:),intent(inout)::coord,coord0
  real(dp),dimension(:,:),intent(out)::vel,force
  real(dp),intent(out)::pener

  ! get potential and force
  call pot_force(coord,force,pener)
  !     update positions using the verlet algorithm
  do i=1,npart
     r(:) = 2*coord(i,:) - coord0(i,:) + force(i,:)*tstep*tstep
     vel(i,:) = (r(:) - coord0(i,:)) / (2*tstep)
     !     periodic boundary conditions
     where ( r > boxl )
        r = r - boxl
     end where
     where ( r < 0d0 )
        r = r + boxl
     end where
     !  update coordinates
     coord0(i,:) = coord(i,:)
     coord(i,:) = r(:)
  enddo
  !   output coordinates

contains
```

```
  subroutine pot_force(coord,force,pener)
    use lennjones

    real(dp) :: f(3),r(3),V

    pener = 0.d0
    force = 0d0

    do i=1,npart-1
       do j=i+1,npart
          r(:) = coord(i,:) - coord(j,:)
          !     periodic boundary conditions
          r(:) = r(:) - nint(r(:)/boxl(:))*boxl(:)
          !     calculate lennard-jones forces and energies
          select case(pot)
          case("lj", "LJ")
             call ljpot(r,f,V)
          case("mp", "MP")
             call morse(r,f,V)
          case default
             call ljpot(r,f,V)
          end select
          pener = pener + V
          force(i,:) = force(i,:) + f(:)
          force(j,:) = force(j,:) - f(:)
       enddo
    enddo

  end subroutine pot_force

end subroutine verlet
```

- Here since **`subroutine`** `pot_force` is an internal procedure, no **`interface`** is required since it is already implicit and all variable declarations are carried over from **`subroutine`** `verlet`

Modules I

- Modules were introduced in Fortran 90 and have a wide range of applications.
- Modules allow the user to write object based code.
- A **`module`** is a program unit whose functionality can be exploited by other programs which attaches to it via the **`use`** statement.
- A **`module`** can contain the following
  1. global object declaration: replaces Fortran 77 COMMON and INCLUDE statements
  2. interface declaration: all external procedures using assumed shape arrrays, intent and keyword/optional arguments must have an explicit interface
  3. procedure declaration: include procedures such as subroutines or functions in modules. Since modules already contain explicit interface, an interface statement is not required

Modules II

```
module precision

  implicit none
  integer,parameter :: dp = selected_real_kind(15)

end module precision

module param
  use precision

  implicit none
  integer :: inp,outp,nstep,istep,i,j,nunit,npart
  real(dp) :: boxl(3),tstep,temp
  character(len=2) :: pot

end module param

module lennjones
  use precision

  implicit none
  real(dp) :: r2,r6,d2,d6

  contains
    subroutine ljpot(r,f,p)
      implicit none
      real(dp),dimension(:),intent(in) :: r
      real(dp),dimension(:),intent(out) :: f
      real(dp),intent(out) :: p

      r2 = 1.0d0 / dot_product(r,r)
      r6 = r2 * r2 * r2

      f(:) = dvdr(r2,r6)*r(:)
      p = epot(r2,r6)
    end subroutine ljpot
```

```
    subroutine morse(r,f,p)
      implicit none
      real(dp),dimension(:),intent(in) :: r
      real(dp),dimension(:),intent(out) :: f
      real(dp),intent(out) :: p

      f(:) = morseforce(dot_product(r,r))*r(:)
      p = morsepot(dot_product(r,r))
    end subroutine morse
    function dvdr(r2,r6)
      implicit none
      real(dp) :: dvdr
      real(dp),intent(in) :: r2,r6
      dvdr = 48*r2*r6*(r6 - 0.5d0)
    end function dvdr

    function epot(r2,r6)
      implicit none
      real(dp) :: epot
      real(dp),intent(in) :: r2,r6
      epot = 4.d0*r6*(r6 - 1.d0)
    end function epot

    function morsepot(d2)
      implicit none
      real(dp),intent(in) :: d2
      real(dp) :: morsepot
      real(dp) :: de,re,a
      de = 0.176d0 ; a = 1.40d0 ; re = 1d0
      morsepot = de * (1d0 - exp(-a*(sqrt(d2)-re)))**2
    end function morsepot

    function morseforce(d2)
      implicit none
      real(dp),intent(in) :: d2
      real(dp) :: morseforce
      real(dp) :: de,re,a,r
      de = 0.176d0 ; a = 1.40d0 ; re = 1d0 ; r = sqrt(d2)
      morseforce = 2d0 * de * a * (1d0 - exp(-a*(r-re)))* &
         exp(-a*(r-re))
    end function morseforce
end module lennjones
```

Modules III

- within a **`module`**, functions and subroutines are called module procedures.
- **`module`** procedures can contain internal procedures
- **`module`** objects that retain their values should be given a **`save`** attribute
- **`module`**s can be used by procedures and other modules, see **`module precision`**.
- **`module`**s can be compiled separately. They should be compiled before the program unit that uses them.

  Observe that in my examples with all code in single file, the **`module`**s appear before the main program and subroutines.

Modules IV

Visibility of module procedures

- By default, all module procedures are public i.e. they can accessed by program units that use the module using the **`use`** statement
- To restrict the visibility of the module procedure only to the module, use the **`private`** statement
- In the **`module lennjones`**, all functions which calculate forces can be declared as private as follows

```
module lennjones
  use precision

  implicit none
  real(dp) :: r2,r6,d2,d6
  public :: ljpot, morse,epot,moresepot
  private :: dvdr, morseforce
  ...
```

- Program Units in the MD code can directly call **`ljpot,morse,epot`** and **`moresepot`** but cannot access **`dvdr`** and **`morseforce`**

Modules V

use statement

- The **use** statement names a module whole public definitions are to be made accessible.

  To use all variables from **module param** in **program md**:

```
program md
  use param
  ...
end program md
```

- **module** entities can be renamed

  To rename **pot** and **tstep** to more user readable variables:

```
use param, pot => potential, tstep => timestep
```

- It's good programming practice to use only those variables from **module**s that are neccessary to avoid name conflicts and overwrite variables.
- For this, use the **use <module name>, only** statement

```
subroutine verlet(coord,force,pener)
  use param,only : dp,npart,boxl,tstep
  ...
end subroutine verlet
```

Compiling Modules I

- Consider the MD code containing a main program `md.f90`, modules `precision.f90`, `param.f90` and `lennjones.f90` and subroutines `init.f90`, `setup.f90`, `verlet.f90`, `rescale.f90`, `gaussran.f90` and `pot_force.f90`.
- In general, the code can be compiled as

```
ifort -o md md.f90 precision.f90 param.f90 lennjones.f90 init.f90 setup.f90 verlet.f90 rescale.f90 gaussran.f90 pot_force.f90
```

- Most compilers are more restrictive in the order in which the modules appear.
- In general, the order in which the sub programs should be compiled is the following
  1. Modules that do not use any other modules.
  2. Modules that use one or more of the modules already compiled.
  3. Repeat the above step until all modules are compiled and all dependencies are resolved.
  4. Main program followed by all subroutines and functions (if any).
- In the MD code, the module `precision` does not depend on any other modules and should be compiled first
- The modules `param` and `lennjones` only depend on `precision` and can be compiled in any order

Compiling Modules II

- The main program and subroutines can then be compiled

```
ifort -o md precision.f90 param.f90 lennjones.f90 md.f90 init.f90 setup.f90 verlet.f90 rescale.f90 gaussran.f90 pot_force.f90
```

- modules are designed to be compiled independently of the main program and create a `.mod` files which need to be linked to the main executable.

```
ifort -c precision.f90 param.f90 lennjones.f90
creates precision.mod param.mod lennjones.mod
```

- The main program can now be compiled as

```
ifort -o md md.f90 init.f90 setup.f90 verlet.f90 rescale.f90 gaussran.f90 pot_force.f90 -I{path to directory containing the .mod files}
```

- The next tutorial on Makefiles will cover this aspect in more detail.

Outline

Derived Types I

- Defined by user (also called structures)
- Can include different intrinsic types and other derived types
- Components are accessed using the percent operator (%)
- Only assignment operator (=) is defined for derived types
- Can (re)define operators - see function overloading
- Derived type definitions should be placed in a **`module`**.
- Previously defined type can be used as components of other derived types.

```
type line_type
  real :: x1, y1, x2, y2
end type line_type
type (line_type) :: a, b
type vector_type
  type(line_type) :: line ! position of center of sphere
  integer :: direction ! 0=no direction, 1=(x1,y1)->(x2,y2) or 2
end type vector_type
type (vector_type) :: c, d
```

Derived Types ***II***

- values can be assigned to derived types in two ways
  1. component by component
     individual component may be selected using the % operator
  2. as an object
     the whole object may be selected and assigned to using a constructor

```
a%x1 = 0.0 ; a%x2 = 0.5 ; a%y1 = 0.0 ; a%y2 = 0.5
c%direction = 0 ; c%line%x1 = 0.0 ; c%line%x2 = 1.0 ; c%line%y1 = -1.0 ; c%line%y2 = 0.0
```

```
b = line_type(0.0, 0.0, 0.5, 0.5)
d%line = line_type(0.0, -1.0, 1.0, 0.0)
d = vector_type( d%line, 1 ) or
d = vector_type( line_type(0.0, -1.0, 1.0, 0.0), 1)
```

- Assigment between two objects of the same derived type is intrinsically defined
  In the previous example: `a = b` is allowed but `a = c` is not.

subroutine setup

```
      coord_t0(pnum+cell)\%x = rcell(cell,1) + real(xx-1,dp)/cells
      coord_t0(pnum+cell)\%y = rcell(cell,2) + real(yy-1,dp)/cells
      coord_t0(pnum+cell)\%z = rcell(cell,3) + real(zz-1,dp)/cells
```

```
      x = rcell(cell,1) + real(xx-1,dp)/cells
      y = rcell(cell,1) + real(yy-1,dp)/cells
      z = rcell(cell,1) + real(zz-1,dp)/cells
      coord_t0(pnum+cell) = dynamics( x, y, z )
```

I/O on Derived Types

- Can do normal I/O on derived types

  `print *, a` will produce the result

  1.0 0.5 1.5

  `print *, c` will produce the result

  2.0 0.0 0.0 0.0

Derived Types IV

Arrays and Derived Types

- Can define derived type objects which contain non-allocatable arrays and arrays of derived type objects

MD code

```
module dynamic_data
  use precision

  implicit none
  type dynamics
     real(dp) :: x,y,z
  end type dynamics

  type(dynamics),dimension(:),allocatable :: coord,vel,force
end module dynamic_data
```

From one of my old codes

```
  type atomic
     character(2)::symbol
     real(dp)::mass,charge,alpc,delc,alpd,alpq,betaone,delta
     integer::number,ls_nprime(2,4),n_shell(0:2),&
          prim_counter(0:3,20),npp(0:3),nls(1:3)
     real(dp)::g_exp(20,0:3,6,20),g_coeff(20,0:3,6,20),&
          g_norm(20,0:3,6,20),sr_coef(0:2,4),&
          sr_exp(0:2,4),ls_coef(2,4),ls_exp(2,4),centerdim
  end type atomic
```

Derived Type Valued Functions

- Functions can return results of an arbitrary defined type.

Derived Types V

Private Derived Types

- A derived type can be wholly private or some of its components hidden

```
module data
  type ::  position
    real, private :: x,y,z
  end type position
  type, private ::  acceleration
    real :: x,y,z
  end type acceleration
  contains
    ...
end module data
```

- Program units that use `data` have `position` exported but not it's components `x,y,z` and the derived type `acceleration`

Pointers I

- In Fortran, a **`pointer`** variable or simply a **`pointer`** is best thought of as a "free-floating" name that may be associated with or "aliased to" some object.
- The object may already have one or more other names or it may be an unnamed object.
- The object represent data (a variable, for example) or be a procedure.
- A **`pointer`** is any variable that has been given the **`pointer`** attribute.
- A variable with the **`pointer`** attribute may be used like any ordinary variable.

Each pointer is in one of the following three states:

undefined	condition of each **`pointer`** at the beginning of a **`program`**, unless it has been initialized
null	not an alias of any data object
associated	it is an alias of some target data object

Pointers II

- **pointer** objects must be declared with the **pointer** attribute

  ```
  real, pointer ::  p
  ```

- Any variable aliased or "pointed to" by a **pointer** must be given the **target** attribute

  ```
  real, target ::  r
  ```

- To make `p` an alias to `r`, use the **pointer assignment statement**

  ```
  p => r
  ```

- The variable declared as a **pointer** may be a simple variable as above, an array or a structure

  ```
  real, dimension(:), pointer :: v
  ```

- **pointer** `v` declared above can now be aliased to a 1D array of reals or a row or column of a multi-dimensional array

  ```
  real, dimension(100,100), target :: a
  v => a(5,:)
  ```

- **pointer** variables can be used as any other variables

  For example, `print *, v` and `print *, a(5,:)` are equivalent

  `v = 0.0` is the same as `a(5,:)  = 0.0'`

Pointers III

- **`pointer`** variables can also be an alias to another **`pointer`** variable

- Consider the following example

```
real, target :: r
real, pointer :: p1, p2
r = 4.7
p1 => r
p2 => r
print *, r, p1, p2
r = 7.4
print *, r, p1, p2
```

- The output on the screen will be

```
4.7 4.7 4.7
7.4 7.4 7.4
```

- Changing the value of `r` to 7.4 causes the value of both `p1` and `p2` to change to 7.4

- Consider the following example

```
real, target :: r1, r2
real, pointer :: p1, p2
r1 = 4.7; r2 = 7.4
p1 => r1 ; p2 => r2
print *, r1, r2, p1, p2
p1 = p2
print *, r1, r2, p1, p2
```

- The output on the screen will be

```
4.7 7.4 4.7 7.4
4.7 4.7 4.7 4.7
```

- The assignment statement `p2 = p1` has the same effect of `r2 = r1` since `p1` is an alias to `r1` and `p2` is an alias to `r2`

Pointers IV

- The **`allocate`** statement can be used to create space for a value and cause a pointer to refer to that space.

  `allocate``(p1)` creates a space for one real number and makes `p1` an alias to that space.

- No real value is stored in that space so it is neccessary to assign a value to `p1`
- `p1 = 4.7` assigns a value 4.7 to that allocated space
- Before a value is assigned to `p1`, it must either be associated with an unnamed target using the **`allocate`** statement or be aliased with a target using the pointer assignment statement.
- **`deallocate`** statement dissociates the pointer from any target and nullifies it

  `deallocate``(p1)`

Pointer Intrinsic Functions

null intrinsic

- **`pointer`** variables are undefined unless they are initialized
- **`pointer`** variable must not be reference to produce a value when it is undefined.
- It is sometime desirable to have a **`pointer`** variable in a state of not pointing to anything
- The **`null`** intrinsic function nullifies a pointer assignment so that it is in a state of not pointing to anything

  `p1 => null()`

- If the target of `p1` and `p2` are the same, then nullifying `p1` does not nullify `p2`
- If `p1` is null and `p2` is pointing to `p1`, then `p2` is also nullified.

associated intrinsic

- The **`associated`** intrinsic function queries whether a pointer varibale is pointing to, or is an alias for another object.

  `associated``(p1,r1)` and **`associated`**`(p2,r2)` are true, but

  `associated``(p1,r2)` and **`associated`**`(p2,r1)` are false

Outline

OOP Concepts

Fortran 90 has some Object Oriented facilites such as

1. data abstraction: user defined types (covered)
2. data hiding - private and public attributes (covered)
3. encapsulation - modules and data hiding facilities (covered)
4. inheritance and extensibility - super-types, operator overloading and generic procedures
5. polymorphism - user can program his/her own polymorphism by generic overloading
6. resuability - modules

Overloading Procedures I

- In Fortran, most intrinsic functions are generic in that their type is determined by their argument(s)
- For example, the **abs** (x) intrinsic function comprises of
  1. **cabs** : called when x is **complex**
  2. **abs** : called when x is **real**
  3. **iabs** : called when x is **integer**
- These sets of functions are called *overload sets*
- Fortran users may define their own *overload sets* in an **interface** block

```
interface clear
   module procedure clear_real, clear_type, clear_type1D
end interface
```

- The generic name `clear` is associated with specific names `clear_real, clear_type, clear_type1D`

Overloading Procedures II

```
module dynamic_data
  ...
  type dynamics
     real(dp) :: x,y,z
  end type dynamics
  interface dot_product
     module procedure dprod
  end interface dot_product
  interface clear
     module procedure clear_real, clear_type, clear_type1D
  end interface
contains
  function dprod(a,b) result(c)
    type(dynamics),intent(in) :: a,b
    real(dp) :: c
    c = a%x * b%x + a%y * b%y + a%z * b%z
  end function dprod
  subroutine clear_real(a)
    real(dp),dimension(:,:),intent(out) :: a
    a = 0d0
  end subroutine clear_real

  subroutine clear_type(a)
    type(dynamics),dimension(:),intent(out) :: a
    a%x = 0d0 ; a%y = 0d0 ; a%z = 0d0
  end subroutine clear_type

  subroutine clear_type1D(a)
    type(dynamics),intent(out) :: a
    a%x = 0d0 ; a%y = 0d0 ; a%z = 0d0
  end subroutine clear_type1D
end module dynamic_data
```

```
program md
  use dynamic_data
  ...
  type(dynamics),dimension(:),allocatable :: coord,coord0,vel,force
  ...
  allocate(coord(npart),coord0(npart),vel(npart),force(npart))
  ...
     do i=1,npart
        v2t = v2t + dot_product(vel(i),vel(i))
     enddo
  ...
end program md

subroutine setup(coord,vel,coord0)
  ...
  type(dynamics) :: vt
  ...
  call clear(coord)
  call clear(coord0)
  call clear(vel)
  ...
  call clear(vt)
  ...
end subroutine setup
```

- The **dot_product** intrinsic function is overloaded to inlcude derived types
- The procedure `clear` is overloaded to set all components of derived types and all elements of 2D real arrays to zero.

Operator Overloading I

- Intrinsic operators such as +, -, * and / can be overloaded to apply to all types of data
- Recall, for derived types only the assignment (=) operator is defined
- In the MD code, `coord_t(i) = coord_t0(i)` is well defined, but
- `vel_t(i) = vel_t(i) * scalef` is not
- Operator overloading as follows
  1. specify the generic operator symbol in an **`interface operator`** statement
  2. specify the overload set in a generic interface
  3. declare the **`module procedure`**s (**`function`**s) which define how the operations are implemented.
  4. these functions must have one or two non-optional arguments with **`intent(in)`** which correspond to monadic or dyadic operators

Operator Overloading II

```
module dynamic_data
  ...
  type dynamics
     real(dp) :: x,y,z
  end type dynamics

  interface operator (*)
     module procedure scale_tr, scale_rt
  end interface operator (*)
  interface operator (+)
     module procedure add
  end interface operator (+)
contains
  type(dynamics) function scale_tr(a,b) result(c)
    type(dynamics),intent(in)::a
    real(dp),intent(in) :: b
    type(dynamics) :: c
    c%x = a%x * b
    c%y = a%y * b
    c%z = a%z * b
  end function scale_tr
  type(dynamics) function scale_rt(b,a) result(c)
    type(dynamics),intent(in)::a
    real(dp),intent(in) :: b
    type(dynamics) :: c
    c%x = b * a%x
    c%y = b * a%y
    c%z = b * a%z
  end function scale_rt
  type(dynamics) function add(a,b) result(c)
    type(dynamics),intent(in) :: a,b
    type(dynamics) :: c
    c%x = a%x + b%x
    c%y = a%y + b%y
    c%z = a%z + b%z
  end function add
end module dynamic_data
```

- The following operator are not defined for derived types `a`, `b`, `c` and scalar `r`

1. `c = a * r`
2. `c = r * a`
3. `c = a + b`

- If operator overloading is not defined, the above operations would have to be executed as follows whereever needed

1. `c%x = a%x * r ; ···`
2. `c%x = r * a%x ; ···`
3. `c%x = a%x + b%x ; ···`

Extended Data Types I

- Recall the derived type example which has as a component another derived type

```
type, public ::  line_type
  real :: x1, y1, x2, y2
end type line_type
type, public ::  vector_type
  type(line_type) :: line ! position of center of sphere
  integer :: direction ! 0=no direction, 1=(x1,y1)->(x2,y2) or 2
end type vector_type
```

- An object, `c`, of type `vector_type` is referenced as `c%line%x1, c%line%y1, c%line%x2, c%line%y2` and `c%direction` which can be cumbersome.
- In Fortran, it is possible to extend the base type `line_type` to other types such as `vector_type` and `painted_line_type` as follows

```
type, public, extends(line_type) ::  vector_type
  integer :: direction
end type vector_type
type, public, extends(line_type) ::  painted_line_type
  integer :: r, g, b ! rgb values
end type painted_line_type
```

Extended Data Types II

- An object,`c` of type `vector_type` inherits the components of the type `line_type` and has components `x1,y1,x2,y2` and `direction` and is reference as `c%x1, c%y1, c%x1, c%y2` and `c%direction`
- Similarly, an object, `d` of type `painted_line_type` is reference as `d%x1, d%y2, d%x2, d%y2, d%r, d%g` and `d%b`
- The three derived types constitute a **`class`**; the name of the class is the name of the base type **`line_type`**

References

- Fortran 95/2003 Explained, Michael Metcalf
- Modern Fortran Explaned, Michael Metcalf
- Guide to Fortran 2003 Programming, Walter S. Brainerd
- Introduction to Programming with Fortran: with coverag of Fortran 90, 95, 2003 and 77, I. D. Chivers
- Fortran 90 course at University of Liverpool, `http://www.liv.ac.uk/HPC/F90page.html`
- Introduction to Modern Fortran, University of Cambridge, `http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/Fortran`

Hands-On Exercise: Molecular Dynamics

- Molecular Dynamics code for melting of solid Hydrogen using Lennard-Jones Potential
- Code can be obtained from QueenBee and Tezpur:
  /work/apacheco/F90-workshop/Exercise/code
- Original (F77) code is in the orig directory.
- Solutions in directories day1, day2 and day3.
- Input file in bench directory, fort.44 and fort.77 are the correct results.
- There is no "correct solution".
- Up to you to decide where you want to finish coding.
- Goal of this Hands-On Exercise should be to use as many features/Concepts of Fortran 90/95 that you have learned and still get the correct result.

Next

Break for Lunch

Make System