Modern Fortran Programming I

Alexander B. Pacheco
LTS Research Computing
April 27, 2015

Tutorial Outline

Day 1: Introduction to Fortran Programming

On the first day, we will provide an introduction to the Fortran 90/95 Programming Language. Go to slide 3

Day 2: Advanced Concepts in Fortran Programming

On the second day, we will cover advanced topics such as modules, derived types, interfaces, etc. Go to slide 104

Part I

Introduction to Fortran Programming

Outline of Part I

Introduction

What is Fortran?

- Fortran is a general-purpose, imperative programming language that is especially suited to numeric computation and scientific computing.
- Originally developed by IBM for scientific and engineering applications.
- The name Fortran is derived from The IBM Mathematical **For**mula **Tran**slating System.
- It was one of the first widely used "high-level" languages, as well as the first programming language to be standardized.
- It is still the premier language for scientific and engineering computing applications.

Many Flavors of Fortran

- FORTRAN — first released by IBM in 1956
- FORTRAN II — released by IBM in 1958
- FORTRAN IV — released in 1962, standardized
- FORTRAN 66 — appeared in 1966 as an ANSI standard
- FORTRAN 77 — appeared in 1977, structured features
- Fortran 90 — 1992 ANSI standard, free form, modules
- Fortran 95 — a few extensions
- Fortran 2003 — object oriented programming
- Fortran 2008 — a few extensions

The correct spelling of Fortran for 1992 ANSI standard and later (sometimes called Modern Fortran) is "Fortran". Older standards are spelled as "FORTRAN".

Why Learn Fortran?

- Fortran was designed by, and for, people who wanted raw number crunching speed.
- There's a great deal of legacy code and numerical libraries written in Fortran,
- attempts to rewrite that code in a more "stylish" language result in programs that just don't run as fast.
- Fortran is the primary language for some of the most intensive supercomputing tasks, such as
  - astronomy,
  - weather and climate modeling,
  - numerical linear algebra and libraries,
  - computational engineering (fluid dynamics),
  - computational science (chemistry, biology, physics),
  - computational economics, etc.
- How many of you are handed down Fortran code that you are expected to further develop?

Why learn Modern Fortran and not FORTRAN?

- FORTRAN is a fixed source format dating back to the use of punch cards.
- The coding style was very restrictive
  - Max 72 columns in a line with
  - first column reserved for comments indicated by a character such as c or *,
  - the second through fifth columns reserved for statement labels,
  - the sixth column for continuation indicator, and
  - columns 7 through 72 for statements.
  - Variable names can consists of up to 6 alphanumeric characters (a-z,0-9)
- Cannot process arrays as a whole, need to do it element by element.
- Cannot allocate memory dynamically.

FORTRAN 77 Example

SAXPY Code

```
C234567890123456789012345678901234567890123456789012345678901234567890
      program test
      integer n
      parameter(n=100)
      real alpha, x(n), y(n)

      alpha = 2.0
      do 10 i = 1,n
         x(i) = 1.0
         y(i) = 2.0
 10   continue

      call saxpy(n,alpha,x,y)

      return
      end

      subroutine saxpy(n, alpha, x, y)
      integer n
      real alpha, x(*), y(*)
c
c Saxpy: Compute y := alpha*x + y,
c where x and y are vectors of length n (at least).
c
      do 20 i = 1, n
         y(i) = alpha*x(i) + y(i)
 20   continue

      return
      end
```

Why Learn Modern Fortran?

- Free-format source code with a maximum of 132 characters per line,
- Variable names can consists of up to 31 alphanumeric characters (a-z,0-9) and underscores (_),
- Dynamic memory allocation and Ability to operate on arrays (or array sections) as a whole,
- generic names for procedures, optional arguments, calls with keywords, and many other procedure call options,
- Recursive procedures and Operator overloading,
- Structured data or derived types,
- Object Oriented Programming.
- See `http://en.wikipedia.org/wiki/Fortran#Obsolescence_and_deletions` for obsolete and deleted FORTRAN 77 features in newer standards.

FORTRAN 90 Example

SAXPY Code

```
program test

  implicit none
  integer, parameter :: n = 100
  real :: alpha, x(n), y(n)

  alpha = 2.0
  x = 1.0
  y = 2.0

  call saxpy(n,alpha,x,y)

end program test

subroutine saxpy(n, alpha, x, y)
  implicit none
  integer ::  n
  real :: alpha, x(*), y(*)
!
! Saxpy: Compute y := alpha*x + y,
! where x and y are vectors of length n (at least).
!
  y(1:n) = alpha*x(1:n) + y(1:n)

end subroutine saxpy
```

Major Differences with C

- **No standard libraries:** No specific libraries have to be loaded explicitly for I/O and math.
- **Implicit type declaration:** In Fortran, variables of type real and integer may be declared implicitly, based on their first letter. *This behaviour is not recommended in Modern Fortran*.
- **Arrays vs Pointers:** Multi-dimension arrays are supported (arrays in C are one-dimensional) and therefore no vector or array of pointers to rows of a matrices have to be constructed.
- **Call by reference:** Parameters in function and subroutine calls are all passed by reference. When a variable from the parameter list is manipulated, the data stored at that address is changed, not the address itself. Therefore there is no reason for referencing and de-referencing of addresses (as commonly seen in C).

Basics

Fortran Source Code I

- Fortran source code is in ASCII text and can be written in any plain-text editor such as vi, emacs, etc.
- For readability and visualization use a text editor capable of syntax highlighting and source code indentation.
- Fortran source code is case insensitive i.e. PROGRAM is the same as Program.
- Using mixed case for statements and variables is not considered a good programming practice. Be considerate to your collaborators who will be modifying the code.
- Some Programmers use uppercase letters for Fortran keywords with rest of the code in lowercase while others (like me) only use lower case letters.
- Use whatever convention you are comfortable with and be consistent throughout.
- The general structure of a Fortran program is as follows

Fortran Source Code II

```
PROGRAM name
   IMPLICIT NONE
   [specification part]
   [execution part]
   [subprogram part]
END PROGRAM name
```

1. A Fortran program starts with the keyword **PROGRAM** followed by program name,
2. This is followed by the **IMPLICIT NONE** statement (avoid use of implicit type declaration in Fortran 90),
3. Followed by specification statements for various type declarations,
4. Followed by the actual execution statements for the program,
5. Any optional subprogram, and lastly
6. The **END PROGRAM** statement

Fortran Source Code III

- A Fortran program consists of one or more program units.
  - **PROGRAM**
  - **SUBROUTINE**
  - **FUNCTION**
  - **MODULE**
- The unit containing the **PROGRAM** attribute is often called the *main program* or *main*.
- The main program should begin with the **PROGRAM** keyword. This is however not required, but it's use if highly recommended.
- A Fortran program should contain only one main program i.e. one **PROGRAM** keyword and can contain one or more subprogram units such as **SUBROUTINE**, **FUNCTION** and **MODULE**.
- Every program unit, must end with a **END** keyword.

Simple I/O

- Any program needs to be able to read input and write output to be useful and portable.
- In Fortran, the **print** command provides the most simple form of writing to standard output while,
- the **read** command provides the most simple form of reading input from standard input
- `print *, <var1> [, <var2> [, ... ]]`
- `read *, <var1> [, <var2> [, ... ]]`
- The $*$ indicates that the format of data read/written is unformatted.
- In later sections, we will cover how to read/write formatted data and file operations.
- variables to be read or written should be separated by a comma (,).

Your first code in Fortran

- Open a text editor and create a file helloworld.f90 containing the following lines

```
program hello
  print *, 'Hello World!'
end program hello
```

- The standard extension for Fortran source files is .f90, i.e., the source files are named <name>.f90.
- The .f extension implies fixed format source or FORTRAN 77 code.

Compiling Fortran Code

- To execute a Fortran program, you need to compile it to obtain an executable.
- Almost all *NIX system come with GCC compiler installed. You might need to install the Fortran (gfortran) compiler if its not present.
- Command to compile a fortran program

```
<compiler> [flags] [-o executable] <source code>
```

- The [...] is optional. If you do not specify an executable, then the default executable is `a.out`

```
altair:Exercise apacheco$ gfortran helloworld.f90
altair:Exercise apacheco$ ./a.out
 Hello World!
```

- Other compilers available on our clusters are Intel (ifort), Portland Group (pgf90) and IBM XL (xlf90) compilers.

```
ifort -o helloworld helloworld.f90; ./helloworld
```

Comments

- To improve readability of the code, comments should be used liberally.
- A comment is identified by an exclamation mark or bang (!), except in a character string.
- All characters after ! upto the end of line is a comment.
- Comments can be inline and should not have any Fortran statements following it

```
program hello
! A simple Hello World code
  print *, 'Hello World!' ! Print Hello World to screen

! This is an incorrect comment if you want Hello World to print to screen ! print
      *, 'Hello World!'
end program hello
```

Variables I

- Variables are the fundamental building blocks of any program.
- In Fortran, a variable name may consist of up to 31 alphanumeric characters and underscores, of which the first character must be a letter.
- There are no reserved words in Fortran.
- However, names must begin with a letter and should not contain a space.
- Allowed names: a, compute_force, qed123
- Invalid names: 1a, a thing, $sign

Variables II

Variable Types

- Fortran provides five intrinsic data types

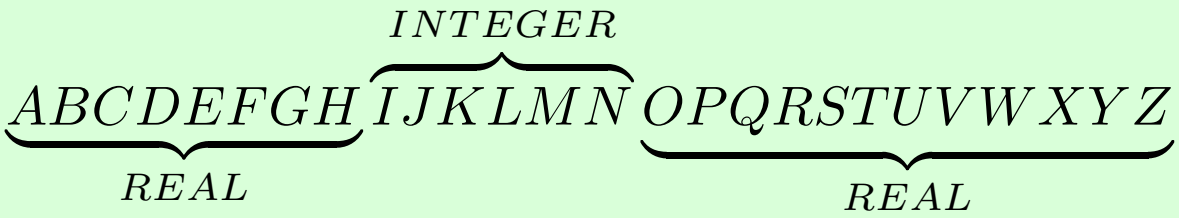
  `INTEGER`: exact whole numbers
  `REAL`: real, fractional numbers
  `COMPLEX`: complex, fractional numbers
  `LOGICAL`: boolean values
  `CHARACTER`: strings

- and allows users to define additional types.
- The **`REAL`** type is a single-precision floating-point number.
- The **`COMPLEX`** type consists of two reals (most compilers also provide a **`DOUBLE COMPLEX`** type).
- FORTRAN also provides **`DOUBLE PRECISION`** data type for double precision **`REAL`**. This is obsolete but is still found in several programs.

Variables III

Explicit and Implicit Typing

- For historical reasons, Fortran is capable of implicit typing of variables.



- You might come across old FORTRAN program containing `IMPLICIT REAL*8(a-h,o-z)` or `IMPLICIT DOUBLE PRECISION (a-h,o-z)`.
- It is highly recommended to explicitly declare all variable and avoid implict typing using the statement `IMPLICIT NONE`.
- The `IMPLICIT` statement must precede all variable declarations.

Literal Constants I

- Constants are literal representation of a type that are specified by a programmer.
- In Fortran, different constant formats are required for each type.

`Integer`

  - Integers are specified by a number without a decimal point,
  - may contain an optional sign, and
  - not contain commas
  - Example

```
137
-5678
900123
```

Literal Constants II

Real

- Reals (single precision) may be specified by adding a decimal point, or by using scientific notation with the letter *e* indicating the exponent.
- Examples

```
19.
3.14159
6.023e23
```

- Double precision constants must be specified in the expontent notation with the letter *d* indicating the exponent
- Examples

```
23d0
6.626d-34
-3.14159d0
```

Literal Constants III

Complex

- Complex constants consist of two single-precision constants enclosed in parenthesis.
- The first constant is the real part; the second is the imaginary part.
- Example

```
(1.0,0.0)
(-2.7e4,5.0)
```

- For systems that support double precision complex, the floating point constants must use the *d* notation.

```
(-2.7d-4,5.d0)
```

Logical

- Logical constants can take only the value `.True.` or `.FALSE.` with the periods part of the constant.

Literal Constants IV

Character

- Character constants are specified by enclosing them in single quotes.
- Example

```
'This is a character constant'
'The value of pi is 3.1415'
```

- If an apostrophe is to be part of the constant, it should be represented by a double quote

```
'All the world''s a stage'
```

Variable Declarations I

- Variables must be declared before they can be used.
- In Fortran, variable declarations must precede all executable statements.
- To declare a variable, preface its name by its type.

```
TYPE Variable
```

- A double colon may follow the type.

```
TYPE[, attributes] :: Variable
```

- This is the new form and is recommended for all declarations. If attributes need to be added to the type, the double colon format must be used.
- A variable can be assigned a value at its declaration.

Variable Declarations II

- **Numeric Variables:**

```
INTEGER :: i, j = 2
REAL    :: a, b = 4.d0
COMPLEX :: x, y
```

- In the above examples, the value of j and b are set at compile time and can be changed later.
- If you want the assigned value to be constant that cannot change subsequently, add the attribute **PARAMETER**

```
INTEGER, PARAMETER :: j = 2
REAL, PARAMETER    :: pi = 3.14159265
COMPLEX, PARAMETER :: ci = (0.d0,1.d0)
```

- **Logical:** Logical variables are declared with the **LOGICAL** keyword

```
LOGICAL  :: l, flag=.true.
```

- **Character:** Character variables are declared with the **CHARACTER** type; the length is supplied via the keyword **LEN**.

Variable Declarations III

- The length is the maximum number of characters (including space) that will be stored in the character variable.
- If the **LEN** keyword is not specified, then by default **LEN**=1 and only the first character is saved in memory.

```
CHARACTER          :: ans = 'yes' ! stored as y not yes
CHARACTER(LEN=10)  :: a
```

- FORTRAN programmers: avoid the use of **CHARACTER***10 notation.

Array Variables

- Arrays (or matrices) hold a collection of different values at the same time.
- Individual elements are accessed by subscripting the array.
- Arrays are declared by adding the **DIMENSION** attribute to the variable type declaration which can be integer, real, complex or character.
- Usage: `TYPE, DIMENSION(lbound:ubound):: variable_name`

  Lower bounds of one can be omitted

```
INTEGER, DIMENSION(1:106) :: atomic_number
REAL, DIMENSION(3, 0:5, -10:10) :: values
CHARACTER(LEN=3), DIMENSION(12) :: months
```

- In Fortran, arrays can have upto seven dimension.
- In contrast to C/C++, Fortran arrays are column major.
- We'll discuss arrays in more details in the Advanced Concepts Tutorials.

DATA Statments

- In FORTRAN, a **DATA** statement may be used to initialize a variable or group of variables.
- It causes the compiler to load the initial values into the variables at compile time i.e. a nonexecutable statment
- General form

```
DATA varlist /varlist/ [, varlist /varlist/]
```

  Example `DATA a,b,c /1.,2.,3./`

- **DATA** statements can be used in Fortran but it is recommended to to eliminate this statement by initializing variables in their declarations.
- In Fortran 2003, variables may be initialized with intrinsic functions (some compilers enable this in Fortran 95)

```
REAL, PARAMETER :: pi = 4.0*atan(1.0)
```

KIND Parameter I

- In FORTRAN, types could be specified with the number of bytes to be used for storing the value:
  - `real*4` - uses 4 bytes, roughly $\pm 10^{-38}$ to $\pm 10^{38}$.
  - `real*8` - uses 8 bytes, roughly $\pm 10^{-308}$ to $\pm 10^{308}$.
  - `complex*16` - uses 16 bytes, which is two `real*8` numbers.
- Fortran 90 introduced `kind` parameters 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)

KIND Parameter II

KIND	Size (Bytes)	Data Type
1	1	integer, logical, character (default)
2	2	integer, logical
4[a]	4	integer, real, logical, complex
8	8	integer, real, logical, complex
16	16	real, complex

[a] default for all data types except character

- You might come across FORTRAN codes with variable declarations using `integer*4`, `real*8` and `complex*16` corresponding to `kind=4` (integer) and `kind=8` (real and complex).
- 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 III

- **selected_int_kind**(R) returns the kind value of the smallest integer type that can represent all values ranging from -10^R (exclusive) to 10^R (exclusive)
- **selected_real_kind**(P,R) returns the kind value of a real data type with decimal precision of at least P digits, exponent range of at least R. At least one of P and R must be specified, default R is 308.

```
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),&
```

KIND Parameter IV

```
      '  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)
 print *, huge(i),kind(i)
 print *, huge(j),kind(j)
 print *, huge(k),kind(k)

end program kind_function
```

```
[apacheco@qb4 Exercise] ./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
```

Operators and Expressions

Fortran defines a number of operations on each data type.

Arithmetic Operators

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

Relational Operators (FORTRAN versions)

- `==` : equal to (.eq.)
- `/=` : not equal to (.ne.)
- `<` : less than (.lt.)
- `<=` : less than or equal to (.le.)
- `>` : greater than (.gt.)
- `>=` : greater than or equal to (.ge.)

Logical Expressions

- .AND. intersection
- .OR. union
- .NOT. negation
- .EQV. logical equivalence
- .NEQV. exclusive or

Character Operators

- `//` : concatenation

Operator Evaluations I

- In Fortran, all operator evaluations on variables is carried out from left-to-right.
- Arithmetic operators have a highest precedence while logical operators have the lowest precedence
- The order of operator precedence can be changed using parenthesis, '(' and ')'
- In Fortran, a user can define his/her own operators.
- User defined monadic operator has a higher precedence than arithmetic operators, while
- dyadic operators has a lowest precedence than logical operators.

Operator Evaluations II

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

Expressions

- An expression is a combination of one or more operands, zero or more operators, and zero or more pairs of parentheses.
- There are three kinds of expressions:
  - An arithmetic expression evaluates to a single arithmetic value.
  - A character expression evaluates to a single value of type character.
  - A logical or relational expression evaluates to a single logical value.
- Examples:

```
x + 1.0
97.4d0
sin(y)
x*aimag(cos(z+w))
a .and. b
'AB' // 'wxy'
```

Statements I

- A statement is a complete instruction.
- Statements may be classified into two types: executable and non-executable.
- Non-executable statements are those that the compiler uses to determine various fixed parameters such as module use statements, variable declarations, function interfaces, and data loaded at compile time.
- Executable statements are those which are executed at runtime.
- A statements is normally terminated by the end-of-line marker.
- If a statement is too long, it may be continued by the ending the line with an ampersand (&).
- Max number of characters (including spaces) in a line is 132 though it's standard practice to have a line with up to 80 characters. This makes it easier for file editors to display code or print code on paper for reading.
- Multiple statements can be written on the same line provided the statements are separated by a semicolon.

Statements II

- Examples:

```
force = 0d0 ; pener = 0d0
do k = 1, 3
   r(k) = coord(i,k) - coord(j,k)
```

- Assignment statements assign an expression to a quantity using the equals sign (=)
- The left hand side of the assignment statement must contain a single variable.
- $x + 1.0 = y$ is not a valid assignment statement.

Intrinsic Functions

- Fortran provide a large set of intrinsic functions to implement a wide range of mathematical operations.
- In FORTRAN code, you may come across intrinsic functions which are prefixed with `i` for integer variables, `d` for double precision, `c` for complex single precision and `cd` for complex double precision variables.
- In Modern Fortran, these functions are overloaded, i.e. they can carry out different operations depending on the data type.
- For example: the **`abs`** function equates to $\sqrt{a^2}$ for integer and real numbers and $\sqrt{\Re^2 + \Im^2}$ for complex numbers.

Arithmetic Functions

Function	Action	Example
INT	conversion to integer	J=INT(X)
REAL	conversion to real	X=REAL(J)
	return real part of complex number	X=REAL(Z)
DBLE[a]	convert to double precision	X=DBLE(J)
CMPLX	conversion to complex	A=CMPLX(X[,Y])
AIMAG	return imaginary part of complex number	Y=AIMAG(Z)
ABS	absolute value	Y=ABS(X)
MOD	remainder when I divided by J	K=MOD(I,J)
CEILING	smallest integer $\geq$ to argument	I=CEILING(a)
FLOOR	largest integer $\leq$ to argument	I=FLOOR(a)
MAX	maximum of list of arguments	A=MAX(C,D)
MIN	minimum of list of arguments	A=MIN(C,D)
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)

[a] use real(x,kind=8) instead

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)
SINH	hyperbolic sine	X=SINH(Y)
COSH	hyperbolic cosine	X=COSH(Y)
TANH	hyperbolic tangent	X=TANH(Y)

hyperbolic functions are not defined for complex argument

Character Functions

len(c)	length
len_trim(c)	length of c if it were trimmed
lge(s1,s2)	returns .true. if s1 follows or is equal to s2 in lexical order
lgt(s1,s2)	returns .true. if s1 follows s1 in lexical order
lle(s1,s2)	returns .true. if s2 follows or is equal to s1 in lexical order
llt(s1,s2)	returns .true. if s2 follows s1 in lexical order
adjustl(s)	returns string with leading blanks removed and same number of trailing blanks added
adjustr(s)	returns string with trailing blanks removed and same number of leading blanks added
repeat(s,n)	concatenates string s to itself n times
scan(s,c)	returns the integer starting position of string c within string s
trim(c)	trim trailing blanks from c

Array Intrinsic Functions

size(x[,n])	The size of x (along the n^{th} dimension, optional)
sum(x[,n])	The sum of all elements of x (along the n^{th} dimension, optional) $sum(x) = \sum_{i,j,k,\ldots} x_{i,j,k,\ldots}$
product(x[,n])	The product of all elements of x (along the n^{th} dimension, optional) $prod(x) = \prod_{i,j,k,\ldots} x_{i,j,k,\ldots}$
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: $a + \imath b \Rightarrow a - \imath b$

Program Structure

Program Structure I

```
PROGRAM program-name
   IMPLICIT NONE
   [specification part]
   [execution part]
   [subprogram part]
END PROGRAM program-name
```

- All Fortran statements are case insensitive.
- Most programmers use lower case letters with upper case letters reserved for program keywords.
- Are you a FORTRAN 77 or older programmer?
  - Use the **IMPLICIT NONE** statement, avoid **implicit real**`*8(a-h,o-z)` statement. Get in the habit of declaring all variables.

Simple Temperature Conversion Problem

- Write a simple program that
  1. Converts temperature from celsius to fahrenheit
  2. Converts temperature from fahrenheit to celsius

```
program temp

  implicit none
  real :: tempC, tempF

  ! Convert 10C to fahrenheit

  tempF = 9 / 5 * 10 + 32

  ! Convert 40F to celsius

  tempC = 5 / 9 * (40 - 32 )

  print *, '10C = ', tempF, 'F'
  print *, '40F = ', tempC, 'C'

end program temp
```

```
altair:Exercise apacheco$ gfortran simple.f90
altair:Exercise apacheco$ ./a.out
 10C =    42.0000000     F
 40F =    0.00000000     C
```

- So what went wrong? $10C = 50F$ and $40F = 4.4C$

Type Conversion I

- In computer programming, operations on variables and constants return a result of the same type.
- In the temperature code, $9/5 = 1$ and $5/9 = 0$. Division between integers is an integer with the fractional part truncated.
- In the case of operations between mixed variable types, the variable with lower rank is promoted to the highest rank type.

Variable 1	Variable 2	Result
Integer	Real	Real
Integer	Complex	Complex
Real	Double Precision	Double Precision
Real	Complex	Complex

Type Conversion II

- As a programmer, you need to make sure that the expressions take type conversion into account

```
program temp

  implicit none
  real :: tempC, tempF

  ! Convert 10C to fahrenhiet

  tempF = 9. / 5. * 10 + 32

  ! Convert 40F to celsius

  tempC = 5. / 9. * (40 - 32 )

  print *, '10C = ', tempF, 'F'
  print *, '40F = ', tempC, 'C'

end program temp
```

```
altair:Exercise apacheco$ gfortran temp.f90
altair:Exercise apacheco$ ./a.out
 10C =    50.0000000     F
 40F =    4.44444466     C
```

- The above example is not a good programming practice.
- 10, 40 and 32 should be written as real numbers (10., 40. and 32.) to stay consistent.

Input and Output

Input and Output Descriptors I

- Input and output are accomplished by operations on files.
- Files are identified by some form of file handle, in Fortran called the **unit number**.
- We have already encountered read and write command such as `print *,` and `read *,`
- Alternative commands for read and write are

```
read(unit,*)
```

```
write(unit,*)
```

- There is no comma after the ')'. FORTRAN allowed statements of the form `write(unit,*),` which is not supported on some compilers such as IBM XLF. Please avoid this notation in FORTRAN programs.
- The default unit number 5 is associated with the standard input, and
- unit number 6 is assigned to standard output.
- You can replace `unit` with $\star$ in which case standard input (5) and output (6) file descriptors are used.

Input and Output Descriptors II

- The second $\star$ in **read**/**write** or the one in the **print** */**read** * corresponds to unformatted input/output.
- If I/O is formatted, then $\star$ is replaced with

  fmt=<**format** specifier>

File Operations I

- A file may be opened with the statement

```
OPEN([UNIT=]un, FILE=fname [, options])
```

- Commonly used options for the open statement are:

  IOSTAT=ios: This option returns an integer ios; its value is zero if the statement executed without error, and nonzero if an error occured.

  ERR=label: label is the label of a statement in the same program unit. In the event of an error, execution is transferred to this labelled statement.

  STATUS=istat: This option indicates the type of file to be opened. Possible values are:

  old : the file specified by the file parameter must exist.
  new : the file will be created and must not exist.
  replace : the file will be created if it does not exist or if it exists, the file will be deleted and created i.e. contents overwritten.
  unknown : the file will be created if it doesn't exist or opened if it exists without further processing.
  scratch : file will exist until the termination of the executing program or until a **close** is executed on that unit.

File Operations II

`position`=`todo`: This options specifies the position where the read/write marker should be placed when opened. Possible values are:

rewind : positions the file at its initial point. Convenient for rereading data from file such as input parameters.

append : positions the file just before the endfile record. Convenient while writing to a file that already exists. If the file is `new`, then the position is at its initial point.

File Operations III

- The status of a file may be tested at any point in a program by means of the **INQUIRE** statement.

```
INQUIRE([UNIT=]un, options)
```

OR

```
INQUIRE(FILE=fname, options)
```

- At least one option must be specified. Options include

IOSTAT=ios: Same use as **open** statement.

EXIST=lex: Returns whether the file exists in the logical variable lex

OPENED=Iop: Returns whether the file is open in the logical variable Iop

NUMBER=num: Returns the unit number associated with the file, or -1 if no number is assigned to it. Generally used with the second form of the **INQUIRE** statement.

NAMED=isnamed: Returns whether the file has a name. Generally used with the first form of the **INQUIRE** statement.

NAME=fname: Returns the name of the file in the character variable fname. Used in conjunction with the **NAMED** option.

File Operations IV

READ=rd: Returns a string YES, NO, or UNKNOWN to the character variable rd depending on whether the file is readable. If status cannot be determined, it returns UNKNOWN.

WRITE=wrt: Similar to the **READ** option to test if a file is writable.

READWRITE=rdwrt: Similar to the **READ** option to test if a file is both readable and writeable.

File Operations V

- A file may be closed with the statement

  ```
  CLOSE([UNIT=]un [, options])
  ```

- Commonly used options for the close statement are:

  IOSTAT=ios: Same use as **open** statement.

  ERR=label: Same use as **open** statement.

  STATUS=todo: What actions needs to be performed on the file while closing it. Possible values are

  keep : file will continue to exist after the close statement, default option except for scratch files.
  delete : file will cease to exist after the close statement, default option for scratch files.

Reading and Writing Data I

- The **WRITE** statement is used to write to a file.
- Syntax for writing a list of variable, varlist, to a file associated with unit number un

  WRITE(un, options)varlist
- The most common options for **WRITE** are:

  FMT=label A format statement label specifier.

  You can also specify the exact format to write the data to be discussed in a few slides.

  IOSTAT=ios Returns an integer indicating success or failure; zero if statement executed with no erros and nonzero if an error occured.

  ERR=label The label is a statement label to which the program should jump if an error occurs.
- The **READ** statement is used to read from a file.
- Syntax for reading a list of variable, varlist, to a file associated with unit number un

Reading and Writing Data II

```
READ(un, options)varlist
```

- Options to the **READ** statement are the same as that of the **WRITE** statement with one additional option,

  END=label The label is a statement label to which the program should jump if the end of file is detected.

List-Directed I/O I

- The simplest method of getting data into and out of a program is list-directed I/O.
- The data is read or written as a stream into or from specified variables either from standard input or output or from a file.
- The unit number associate with standard input is 5 while standard output is 6.
- If data is read/written from/to standard input/output, then
  - the unit number, `un` can also be replaced with `*`,
  - use alternate form for reading and writing i.e. the **`read`** `*,` and **`print`** `*,` covered in an earlier slide.
  - If data is unformatted i.e. plain ASCII characters, the option to **`write`** and **`read`** command is `*`
- Example of list-directed output to standard output or to a file associated with unit number 8

```
print *, a, b, c, arr
write(*,*) a, b, arr
write(6,*) a, b, c, arr
write(8,*) a, b, c, &
  arr
```

List-Directed I/O II

- Unlike C/C++, Fortran always writes an end-of-line marker at the end of the list of item for any **print** or **write** statements.
- Printing a long line with many variables may thus require continuations.
- Example of list-directed input from standard output or to a file associated with unit number 8

```
read *, a, b, c, arr
read(*,*) a, b, c, arr
read(5,*) a, b, c, arr
read(8,*) a, b, c, arr
```

- When reading from standard input, the program will wait for a response from the console.
- Unless explicitly told to do so, no prompts to enter data will be printed. Very often programmers use a print statement to let you know that a response is expected.

```
print *, 'Please enter a value for the variable inp'
read *, inp
```

Formatted Input/Output I

- List-directed I/O does not always print the results in a particularly readable form.
- For example, a long list of variable printed to a file or console may be broken up into multiple lines.
- In such cases it is desirable to have more control over the format of the data to be read or written.
- Formatted I/O requires that the programmer control the layout of the data.
- The type of data and the number of characters that each element may occupy must be specified.

Formatted Input/Output II

- A formatted data description must adhere to the generic form,

  `nCw.d`

  where
  - `n` is an integer constant that specifies the number of repititions (default 1 can be omitted),
  - `C` is a letter indicating the type of the data variable to be written or read,
  - `w` is the total number of spaces allocated to this variable, and,
  - `d` is the number of spaces allocated to the fractional part of the variable. Integers are padded with zeros for a total width of `w` provided $d \leq w$.
  - The decimal (.) and `d` designator are not used for integers, characters or logical data types. Note that `d` designator has a different meaning for integers and is usually referred to as `m` to avoid confusion.
- Collectively, these designators are called **edit descriptors**.
- The space occupied by an item of data or variable is called *field*.

Formatted Input/Output III

Data Type	Edit Descriptor	Examples	Result
Integer	nIw[.m]	I5.5	`00010`
Real[a] (floating point)	nFw.d	F12.6	`␣␣␣10.123456`
Real (exponential)	Ew.d[en][b]	E15.8	`␣␣␣0.12345678E1`
Real (engineering)	ESw.d[c]	ES12.3	`␣␣␣50.123E-3`
Character	nAw	A12	`␣␣␣␣␣Fortran`

[a] For complex variables, use two appropriate real edit descriptors

[b] en is used when you need more than 2 digits in the exponent as in 100. E15.7e4 to represent 2.3×10^{1021}

[c] data is printed in multiples of 1000

- **Control descriptors** alter the input or output by addings blanks, new lines and tabs.

Space	nX	add n spaces
Tabs	tn	tab to position n
	tln	tab left n positions
	trn	tab right n positions
New Line	/	Create a new line record

Format Statements I

- Edit descriptors must be used in conjunction with a **PRINT**, **WRITE** or **READ** statement.
- In the simplest form, the format is enclosed in single quotes and parentheses as as argument to the keyword.

```
print '(I5,5F12.6)', i, a, b, c, z ! complex z
write(6,'(2E15.8)') arr1, arr2
read(5,'(2a)') firstname, lastname
```

- If the same format is to be used repeatedly or it is complicated, the **FORMAT** statement can be used.
- The **FORMAT** statement must be labeled and the label is used in the input/output statement to reference it

```
label FORMAT(formlist)
PRINT label, varlist
WRITE(un, label) varlist
READ(un, label) varlist
```

Format Statements II

- The **FORMAT** statements can occur anywhere in the same program unit. Most programmers list all **FORMAT** statements immediately after the type declarations before any executable statements.

```
10 FORMAT(I5,5F12.6)
20 FORMAT(2E15.8)
100 FORMAT(2a)

print 10, i, a, b, c, z ! complex z
write(6,20) arr1, arr2
read(5,100) firstname, lastname
```

Namelist I

- Many scientific codes have a large number of input parameters.
- Remembering which parameter is which and also the order in which they are to read, make creating input files very tedious.
- Fortran provides **NAMELIST** input simplify this situation.
- In a **NAMELIST**, parameters are specified by name and value and can appear in any order.
- The **NAMELIST** is declared as a non-executable statement in the subprogram that reads the input and the variables that can be specified in it are listed.

```
NAMELIST /name/ varlist
```

- Namelists are read with a special form of the **READ** statement.

```
READ(un,[nml=]name)
```

Namelist II

- The input file must follow a particular format:
  - begin with an ampersand followed by the name of the namelist (&name) and ends with a slash (/),
  - variables are specified with an equals sign (=) between the variable name and its value,
  - only statis objects may be part of a namelist; i.e. dynamically allocated arrays, pointers and the like are not permitted
- For example, consider a program that declares a namelist as follows:

```
namelist/moldyn/natom,npartdim,tempK,nstep,dt
```

- The corresponding input file can take the form

```
&moldyn
 npartdim =  10
 tempK = 10d0
 nstep = 1000
 dt = 1d-3
/
```

- Note:
  - parameters may appear in any order in the input file, and
  - may be omitted if they are not needed i.e. they can take default values that is specified in the program

Namelist III

- The above namelist can be read with a single statement as in (other options to `READ` statement can be added if needed)

```
READ(10, nml=moldyn)
```

- To write the values of a namelist is similar

```
WRITE(20, nml=moldyn)
```

- Namelist names and variables are case insensitive.
- The namelist designator cannot have blanks
- Arrays may be namelist variables, but all the values of the array must be listed after the equals sign following its name
- If any variable name is repeated, the final value is taken.
- Namelist are convenient when you want to read different input for different types of calculations within the same program.
- Amber Molecular Dynamics package uses namelist to read input. The following is the input file from Amber's test directory.

Namelist IV

```
&cntrl
  ntx=1, imin=5, ipb=1, inp=2, ntb=0,
/
&pb
  npbverb=0, istrng=0, epsout=80.0, epsin=1.0, space=0.5,
  accept=0.001, sprob=1.6, radiopt=1, dprob=1.6,
/
```

- If multiple variables are listed on the same line, they need to be separated by a comma (,) not semicolon(;)

Internal Read and Write I

- Fortran allows a programmer to cast numeric types to character type and vice versa.
- The character variable functions as an internal file.
- An **internal write** converts from numeric to character type, while
- an **internal read** converts from character to numeric type.
- This is useful feature particularly for writing output of arrays that are dynamically allocated.
- Example: Convert an integer to a character

```
CHARACTER(len=10) :: num
INTEGER           :: inum
WRITE(NUM,'(A10)') inum
```

Internal Read and Write II

- Example: Convert an character to an integer

```
CHARACTER(len=10) :: num = "435"
INTEGER   :: inum
READ(inum,'(I4)') num
```

- Example: Writing data when parameters are not known at compile time

```
CHARACTER(len=23) :: xx
CHARACTER(len=13) :: outfile
INTEGER  :: natoms, istep
REAL     :: time
REAL, ALLOCATABLE, DIMENSION(:) :: coords

natoms = 100 ; ALLOCATE(coords(natoms*3))

WRITE(xx,'(A,I5,A)') '(F12.6,', 3*natoms, '(2X,E15.8))'
WRITE(outfile,'(A8,I5.5,A4)') 'myoutput', istep, '.dat'

OPEN(unit = 10, file = outfile)
WRITE(10, xx) time, coords(:)
```

Control Constructs

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 ( 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 ( expression 1) then
  executable statements
else if ( expression 2 ) then
  executable statements
else if ···

  .
  .
  .
else
  executable statements
end if
```

- Examples:

If-then-else Construct II

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

- 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.

If-then-else Construct III

no `else if`

```
[outer_name:] if ( expression ) then
  executable statements
else
  executable statements
  [inner_name:] if ( expression ) then
    executable statements
  end if [inner_name]
end if [outer_name]
```

no `else`

```
if ( expression ) then
  executable statements
else if ( expression ) then
  executable statements
else if ( expression ) then
  executable 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,f6.3,a,f6.3,a,f6.3,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'
```

Finding roots of quadratic equation II

```
            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
         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
```

Finding roots of quadratic equation III

```
[apacheco@qb4 Exercise] ./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 Exercise] ./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 Exercise] ./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.

```
[case_name:] select case ( expression )
  case ( selector )
    executable statement
  case ( selector )
    executable statement
  case default
    executable statement
end select [case_name]
```

- The 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 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

- Example for character case selector

```
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
```

- Example for integer case selector

```
select case ( score )
  case ( 50 : 59 )
    GRADE = "D"
  case ( 60 : 69 )
    GRADE = "C"
  case ( 70 : 79 )
    GRADE = "B"
  case ( 80 : )
    GRADE = "A"
  case default
    GRADE = "F"
end select
```

Do Construct I

- 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_name:] do loop_control
  execution statements
end do [do_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
- 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*

Do Construct II

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 III

```
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 Exercise] ./fact1
 Enter an integer < 15
10
  10!=        3628800
```

```
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
```

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

Do Construct: Nested I

- 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 and statements following the named loop is executed.
- 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.

Do Construct: Nested II

```
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 = 0,180,45
     inner: do j = 0,180,45
        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 Exercise] ./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

```
[do_name:] do
  executable statements
end do [do_name]
```

- The `executable statements` will be executed indefinitly.
- 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_name:] do while ( expression )
  executable statements
end do [do_name]
```

- 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
```

End of Day 1

- Thats all for Day 1
- Any Question?
- In the second part of the tutorial we will cover advanced topics:
  1. Arrays: Dynamic Arrays, Array Conformation concepts, Array declarations and Operations, etc.
  2. Procedures: Modules, Subroutines, Functions, etc.
  3. Object Oriented Concepts: Derived Type Data, Generic Procedures and Operator Overloading.

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 coverage 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`
- Scientific Programming in Fortran 2003: A tutorial Including Object-Oriented Programming, Katherine Holcomb, University of Virginia.

Exercise

Calculate pi by Numerical Integration I

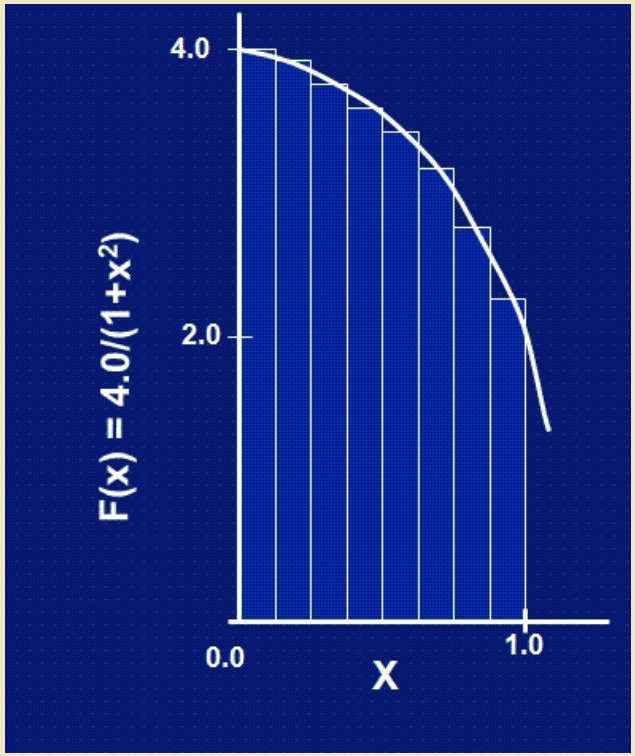
- We know that

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

- So numerically, we can approxiate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A “hands-on” introduction to OpenMP, SC09



Calculate pi by Numerical Integration II

Algorithm 1 Pseudo Code for Calculating Pi

program CALCULATE_PI
  $step \leftarrow 1/n$
  $sum \leftarrow 0$
  do $i \leftarrow 0 \cdots n$
    $x \leftarrow (i + 0.5) * step; sum \leftarrow sum + 4/(1 + x^2)$
  end do
  $pi \leftarrow sum * step$
end program

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines

  $y \leftarrow \alpha x + y$
- Write a SAXPY code to multiply a vector with a scalar.

Algorithm 2 Pseudo Code for SAXPY

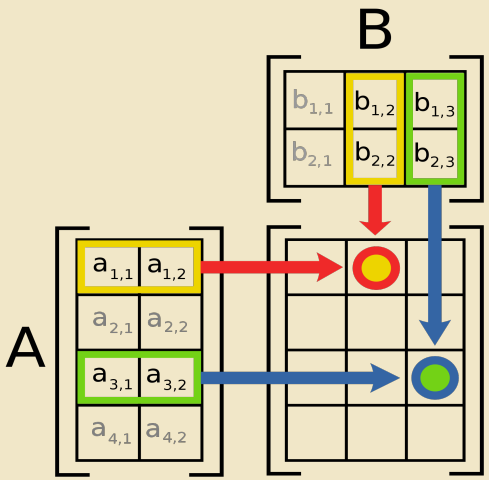
---

program SAXPY
  $n \leftarrow$ some large number
  $x(1:n) \leftarrow$ some number say, 1
  $y(1:n) \leftarrow$ some other number say, 2
  $a \leftarrow$ some other number ,say, 3
  do $i \leftarrow 1 \cdots n$
    $y_i \leftarrow y_i + a * x_i$
  end do
end program SAXPY

Matrix Multiplication I

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix **C** which is a product of two matrices **A** and **B**:

  Element *i,j* of **C** is the dot product of the i^{th} row of **A** and j^{th} column of **B**
- Write a MATMUL code to multiple two matrices.



Matrix Multiplication II

Algorithm 3 Pseudo Code for MATMUL

program MATMUL
  $m, n \leftarrow$ some large number ≤ 1000
  Define a_{mn}, b_{nm}, c_{mm}
  $a_{ij} \leftarrow i + j; b_{ij} \leftarrow i - j; c_{ij} \leftarrow 0$
  do $i \leftarrow 1 \cdots m$
    do $j \leftarrow 1 \cdots m$
      $c_{i,j} \leftarrow \sum_{k=1}^{n} a_{i,k} * b_{k,j}$
    end do
  end do
end program MATMUL

Part II

Advanced Concepts in Fortran Programming

Outline of Part II

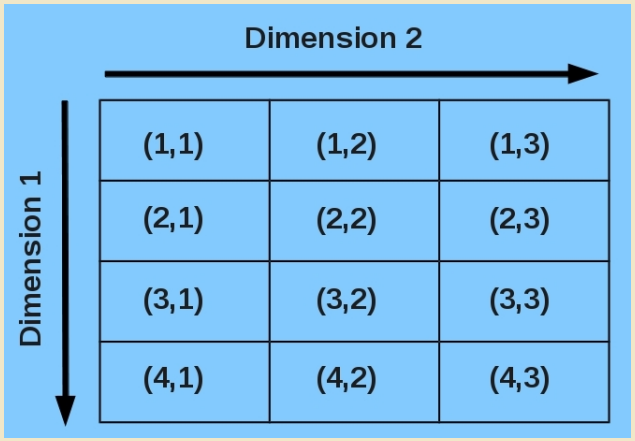
Arrays

Arrays

- 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 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)

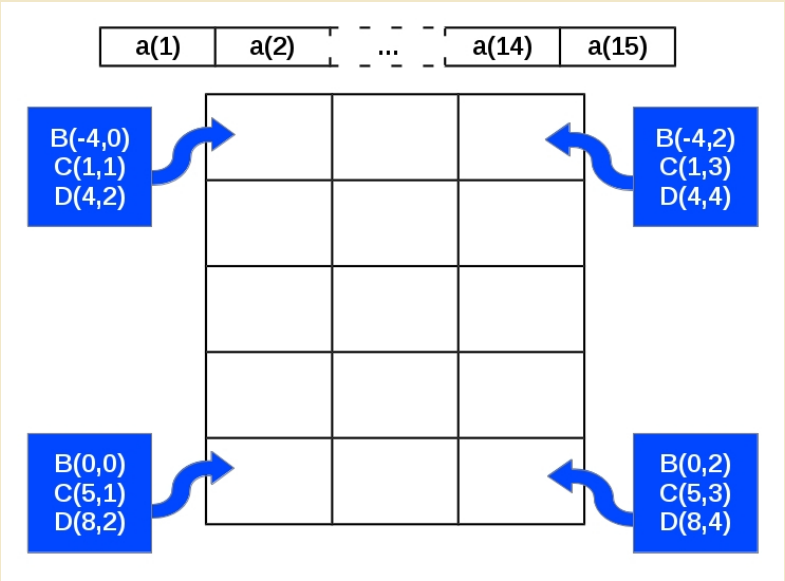
- Arrays are conformable if they share a shape.
- The bounds do not have to be the same

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

Array Visualization

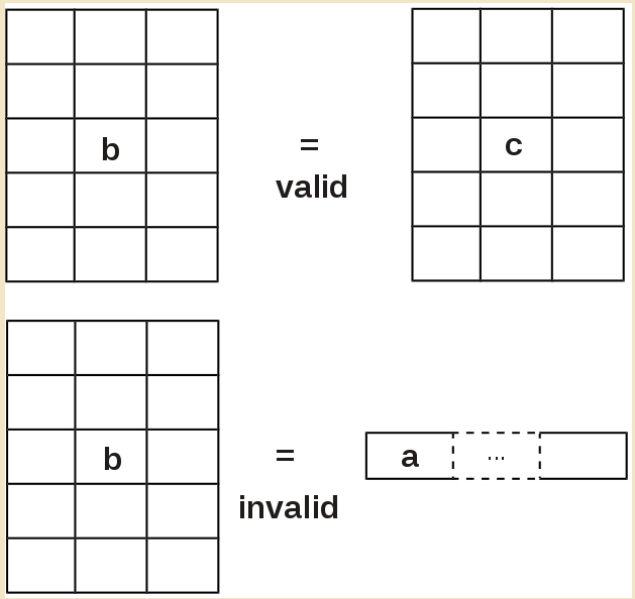
- Define arrays `a`, `b`, `c` and `d` as follows

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



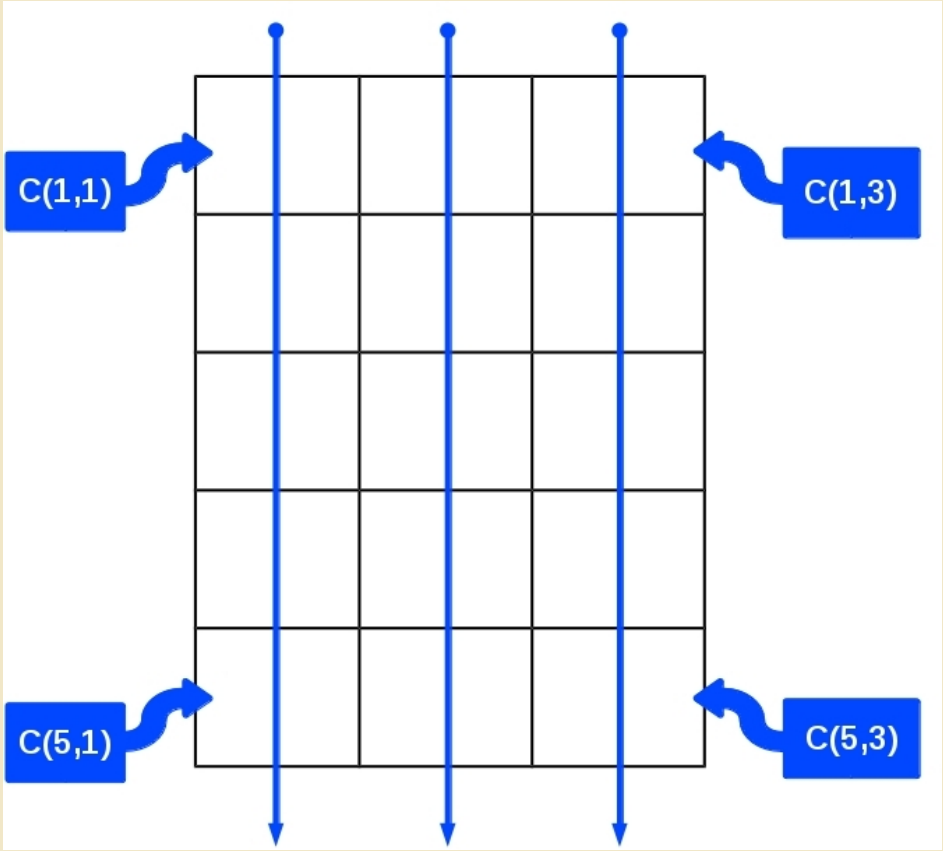
Array Conformance

- 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.



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 Constructors I

- Used to give arrays or sections of arrays specific values

```
implicit none
integer :: i
integer, dimension(10) :: ints
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)

Array Constructors II

- **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 (max value is 7) is the number of elements in **shape**.

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

```
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.

Array Constructors III

- 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++.
- 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

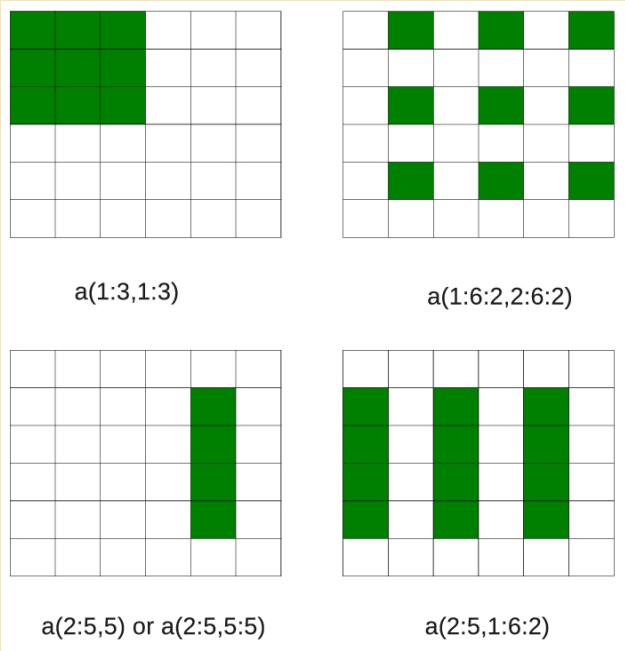
- Arrays can be treated as a single variable when performing operations
  1. set whole array to a constant: `a = 0.0`
  2. can use intrinsic operators between conformable arrays (or sections)
     `b = c * d + b**2`
     this is equivalent to

```
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
```

  3. elemental intrinsic functions can be used: `b = sin(c)+ cos(d)`
  4. All operations/functions are applied element by element

```
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.

```
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**.

- 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
sum(x[,n])	The sum of all elements of x (along the n^{th} dimension, optional)

$sum(x) = \sum_{i,j,k,\ldots} x_{i,j,k,\ldots}$

Array Intrinsic Functions II

product(x[,n]) The product of all elements of x (along the n^{th} dimension, optional)
$prod(x) = \prod_{i,j,k,\ldots} x_{i,j,k,\ldots}$

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: $a + \imath b \Rightarrow a - \imath b$

cshift(ARRAY, SHIFT, dim) perform a circular shift by SHIFT positions to the left on array ARRAY along the $\mathrm{dim}^{\mathrm{th}}$ dimension

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 4000 atoms. If you want to run a simulation for 256 and 1024 atoms, do you need to recompile and create two executables?

- 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))
```

Allocatable Arrays II

- 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)
```

- 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

```
! Apply PBC to coordinates
where ( coord(i,:) > boxl(:) )
   coord(i,:) = coord(i,:) - boxl(:)
elsewhere ( coord(i,:) < 0d0 )
   coord(i,:) = coord(i,:) + boxl(:)
end where
```

```
! Apply PBC to coordinates
do j = 1, 3
   if ( coord(i,j) > boxl(j) ) then
      coord(i,j) = coord(i,j) - boxl(j)
   else if ( coord(i,j) < 0d0 ) then
      coord(i,j) = coord(i,j) + boxl(j)
   endif
end do
```

Procedures

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 ! specify which modules to use

  implicit none  ! implicit typing is not recommended
  variable declarations  ! declare all variables used in the program


  .
  .
  . ! executable statements in seqeunce

  call routine1(arg1,arg2,arg3) ! call subroutine routine1 with arguments

  .
  .
  .
  abc = func(arg1,arg2) ! abc is some function of arg1 and arg2

  .
  .
  .

  contains  ! internal procedures are listed below

    subroutine routine1(arg1,arg2)  ! subroutine routine1 contents go here

      .
      .
      .
    end subroutine routine1  ! all program units must have an end statement

    function func(arg1,arg2) ! function func1 contents go here
      ...
    end function func

  end program main
```

Program Units III

```
program md

  ! Molecular Dynamics code for equilibration of Liquid Argon
  ! Author: Alex Pacheco
  ! Date  : Jan 30, 2014

  ! This program simulates the equilibration of Liquid Argon
  ! starting from a FCC crystal structure using Lennard-Jones
  ! potential and velocity verlet algorithm

  ! This program should be the starting point to learn Modern
  ! Fortran.
  ! This program is hard coded for 4000 atoms equilibrated at
  ! 10K with a time step of 0.001 time units and 1000 time steps
  ! Lets assume that time units is femtoseconds, so total simulation
  ! time is 1 femtosecond

  ! Your objective for
  ! Modern Fortran Training:
  ! Modify this code using the Fortran Concepts learned
  !   1. split code into smaller subunits, modules and/or subroutines
  !   2. generalize, so that the following parameters can be read from a input file
  !       a. number of atoms or number of unit cells (you can't do both)
  !       b. equilibration temperature
  !       c. time step
  !       d. number of time steps i.e. how long in fs do you want the simulation to run
  !       e. read input parameters using namelists
  !      You will need to make use allocatable arrays. If you do not know why, review
  !      training slides or ask
  !   3. Can you use Modern Fortran Concepts such as derived types? If yes, program it
  !   4. If you use derived types, can you overload operators? If yes, program it
  ! OpenMP/OpenACC Training
  !   Lets assume that you have completed upto step 2 from Modern Fortran objective
  !   Parallelize the code for OpenMP/OpenACC (can also be done from step 3 or 4)
  !
  ! There is no time limit for completing this exercise. This exercise is for measuring
  ! what have you got from the training.
  ! Solutions are present in the separate directories for comparison.
  ! Hints are provided whereever needed
```

Program Units IV

```
  ! As an additional exercise, use other potentials such as Morse potential and
  ! read from input file which potential you want to use.
  ! All Lennard-Jones Potential parameters are set to 1.

  ! Disclaimer:
  ! This is code can be used as an introduction to molecular dynamics. There are lot more
  ! concepts in MD that are not covered here.

  ! Parameters:
  ! npartdim : number of unit cells, uniform in all directions. change to nonuniform if you desire
  ! natom : number of atoms
  ! nstep : nummber of simulation time steps
  ! tempK : equilibration temperature
  ! dt : simulation time steps
  ! boxl : length of simulation box in all directions
  ! alat : lattice constant for fcc crystal
  ! kb : boltzmann constant, set to 1 for simplicity
  ! mass : mass of Ar atom, set to 1 for simplicity
  ! epsilon, sigma : LJ parameters, set to 1 for simplicity
  ! rcell : FCC unit cell
  ! coord, coord_t0 : nuclear positions for each step, current and initial
  ! vel, vel_t0 : nuclear velocities for each step
  ! acc, acc_t0 : nuclear acceleration for each step
  ! force, pener : force and potential energy at current step
  ! avtemp : average temperature at current time step
  ! scale : scaling factor to set current temperature to desired temperature
  ! gasdev : Returns a normally distributed deviate with zero mean and unit variance from Numerical recipes

    implicit none
    ! Use either kind function or selected_real_kind
    integer,parameter :: npartdim = 10
    integer,parameter :: natom = 4.d0 * npartdim ** 3
    integer,parameter :: nstep = 1000
    real*8, parameter :: tempK = 10, dt = 1d-3
    integer :: istep
    real*8 :: boxl(3), alat
    integer :: n, i, j, k, l

  ! Can you use derived types for coord, vel, acc and force
    real*8 :: coord_t0(natom,3), coord(natom,3)
```

Program Units V

```
real*8 :: vel_t0(natom,3), vel(natom,3)
real*8 :: acc_t0(natom,3), acc(natom,3)
real*8 :: force(natom,3), pener, mass

real*8 :: vcm(3), r(3), rr, r2, r6, f
real*8 :: avtemp, ke, kb, epsilon, sigma, rcell(3,4), scale
real*8 :: gasdev

alat = 2d0 ** (2d0/3d0)
! Hint: Array operations
do i = 1, 3
   boxl(i) = npartdim * alat
end do
kb = 1.d0
mass = 1.d0
epsilon = 1.d0
sigma = 1.d0

! Create FCC unit cell
! Hint: Simplify unit cell creation, maybe in variable declaration
rcell(1,1) = 0d0
rcell(2,1) = 0d0
rcell(3,1) = 0d0
rcell(1,2) = 0.5d0 * alat
rcell(2,2) = 0.5d0 * alat
rcell(3,2) = 0d0
rcell(1,3) = 0d0
rcell(2,3) = 0.5d0 * alat
rcell(3,3) = 0.5d0 * alat
rcell(1,4) = 0.5d0 * alat
rcell(2,4) = 0d0
rcell(3,4) = 0.5d0 * alat

! Set initial coordinates, velocity and acceleration to zero
! Hint: Use Array operations
do i = 1, natom
   do j = 1, 3
      coord_t0(i,j) = 0d0
      vel_t0(i,j) = 0d0
      acc_t0(i,j) =  0d0
```

Program Units VI

```
   end do
end do

!================================================
! Initialize coordinates and random velocities
!================================================

! Put initialization in a seperate subroutine
! call initialize(coord_t0, vel_t0, ...)
! Create a FCC crystal structure
n = 1
do i = 1, npartdim
   do j = 1, npartdim
      do k = 1, npartdim
         do l = 1, 4
            coord_t0(n,1) = alat * dble(i - 1) + rcell(1,l)
            coord_t0(n,2) = alat * dble(j - 1) + rcell(2,l)
            coord_t0(n,3) = alat * dble(k - 1) + rcell(3,l)
            n = n + 1
         end do
      end do
   end do
end do

open(unit=1,file='atom.xyz',status='unknown')
write(1,'(i8)') natom
write(1,*)
do i = 1, natom
   write(1,'(a2,2x,3f12.6)') 'Ar', coord_t0(i,1), coord_t0(i,2), coord_t0(i,3)
end do
close(1)

! Assign initial random velocities
do i = 1, natom
   do j = 1, 3
      vel_t0(i,j) = gasdev()
   end do
end do

! Set Linear Momentum to zero
```

Program Units VII

```
! Hint: This is needed again below so put in a subroutine
! call linearmom(vel_t0, ...)
! First get center of mass velocity
vcm = 0d0
do i = 1, natom
   do j = 1, 3
      vcm(j) = vcm(j) + vel_t0(i,j)/natom
   end do
end do
! Now remove center of mass velocity from all atoms
do i = 1, natom
   do j = 1, 3
      vel_t0(i,j) = vel_t0(i,j) - vcm(j)
   end do
end do

! scale velocity to desired tempearture
! call get_temp( vel_t0, ... ) will be needed again
ke = 0d0
do i = 1, natom
   do j = 1, 3
      ! Hint: Use dot_product function to calculate vel**2
      ! If using derived types, overload dot_product function
      ke = ke + mass * vel_t0(i,j)**2
   end do
end do
avtemp = mass * ke / ( 3d0 * kb * ( natom - 1))

print '(a,2x,1pe15.8)', 'Initial Average Temperature: ', avtemp

! scale initial velocity to desired temperature
scale = sqrt( tempK / avtemp )
ke = 0d0
do i = 1, natom
   do j = 1, 3
      vel_t0(i,j) = vel_t0(i,j) * scale
      ! See Hint above on dot_product and function overloading
      ke = ke + mass * vel_t0(i,j)**2
   end do
end do
```

Program Units VIII

```
avtemp = mass * ke / ( 3d0 * kb * ( natom - 1))
print '(a,2x,1pe15.8)', 'Initial Scaled Average Temperature: ', avtemp


!================================================
! MD Simulation
!================================================

do istep = 1, nstep

   ! Set coordinates, velocity, acceleration and force at next time step to zero
   ! Hint: Use Array properties
   do i = 1, natom
      do j = 1, 3
         coord(i,j) = 0d0
         vel(i,j) = 0d0
         acc(i,j) =  0d0
         force(i,j) =  0d0
      end do
   end do
   pener = 0d0

   ! Get new atom positions from Velocity Verlet Algorithm
   ! Hint: create a subroutine to do velocity verlet
   ! Hint: OpenMP/OpenACC
   do i = 1, natom
      do j = 1, 3
         coord(i,j) = coord_t0(i,j) + vel_t0(i,j) * dt + 0.5d0 * acc_t0(i,j) * dt ** 2
         ! Apply PBC to coordinates
         if ( coord(i,j) > boxl(j) ) then
            coord(i,j) = coord(i,j) - boxl(j)
         else if ( coord(i,j) < 0d0 ) then
            coord(i,j) = coord(i,j) + boxl(j)
         endif
      end do
   end do

   ! Get force at new atom positions
   ! Using Lennard Jones Potential
   ! Hint: you might want to also seperate the potential and force calculation into a separate subroutine
```

Program Units IX

```
! this will be useful if you want to use other potentials

do i = 1, natom - 1
   do j = i + 1, natom
      do k = 1, 3
         r(k) = coord(i,k) - coord(j,k)
         ! minimum image criterion
         ! interaction of an atom with another atom or its image within the unit cell
         r(k) = r(k) - nint( r(k) / boxl(k) ) * boxl(k)
      end do
      ! Hint: Use dot_product
      rr = r(1) ** 2 + r(2) ** 2 + r(3) ** 2
      r2 = 1.d0 / rr
      r6 = r2 ** 3
      ! Lennard Jones Potential
      ! V = 4 * epsilon * [ (sigma/r)**12 - (sigma/r)**6 ]
      !   = 4 * epsilon * (sigma/r)**6 * [ (sigma/r)**2 - 1 ]
      !   = 4 * r**(-6) * [ r**(-2) - 1 ] for epsilon=sigma=1
      ! F_i = 48 * epsilon * (sigma/r)**6 * (1/r**2) * [ ( sigma/r)** 6 - 0.5 ] * i where i = x,y,z
      !     = 48 * r**(-8) * [ r**(-6) - 0.5 ] * i  for epsilon=sigma=1
      pener = pener + 4d0 * r6 * ( r6 - 1.d0 )
      f = 48d0 * r2 * r6 * ( r6 - 0.5d0 )
      do k = 1, 3
         ! use array function to obtain r(k)*f
         force(i,k) = force(i,k) + r(k) * f
         force(j,k) = force(j,k) - r(k) * f
      end do
   end do
end do

! Calculate Acceleration and Velocity  at current time step
do i = 1, natom
   do j = 1, 3
      acc(i,j) = force(i,j) / mass
      vel(i,j) = vel_t0(i,j) + 0.5d0 * (acc(i,j) + acc_t0(i,j)) * dt
   end do
end do

! Set Linear Momentum to zero
! First get center of mass velocity
```

Program Units X

```
! See Hint above on Linear Momentum
vcm = 0d0
do i = 1, natom
   do j = 1, 3
      vcm(j) = vcm(j) + vel(i,j)/natom
   end do
end do
! Now remove center of mass velocity from all atoms
do i = 1, natom
   do j = 1, 3
      vel(i,j) = vel(i,j) - vcm(j)
   end do
end do

! compute average temperature
! See Hint above on calculating average temperature
ke = 0d0
do i = 1, natom
   do j = 1, 3
      ke = ke + vel(i,j) ** 2
   end do
end do
avtemp = mass * ke / ( 3d0 * kb * ( natom - 1))

print '(a,2x,i8,2x,1pe15.8,1x,1pe15.8)', 'Average Temperature: ' , istep, avtemp, pener

scale = sqrt ( tempk/ avtemp )
! Reset for next time step
! Hint:  Use Array properties
do i = 1, natom
   do j = 1, 3
      acc_t0(i,j) = acc(i,j)
      coord_t0(i,j) = coord(i,j)
      ! scale velocity to desired temperature
      vel_t0(i,j) = vel(i,j) * scale
   end do
end do

! Write current coordinates to xyz file for visualization
open(unit=1,file='atom.xyz',position='append')
```

Program Units XI

```
      write(1,'(i8)') natom
      write(1,*)
      do i = 1, natom
         write(1,'(a2,2x,3f12.6)') 'Ar', coord_t0(i,1), coord_t0(i,2), coord_t0(i,3)
      end do
      close(1)
   end do

end program md

double precision function gasdev()
   implicit none
   real*8 :: v1, v2, fac, rsq
   real*8, save :: gset
   logical, save :: available = .false.

   if (available) then
      gasdev = gset
      available = .false.
   else
      do
         call random_number(v1)
         call random_number(v2)
         v1 = 2.d0 * v1 - 1.d0
         v2 = 2.d0 * v2 - 1.d0
         rsq = v1**2 + v2**2
         if ( rsq > 0.d0 .and. rsq < 1.d0 ) exit
      end do
      fac = sqrt(-2.d0 * log(rsq) / rsq)
      gasdev = v1 * fac
      gset = v2 * fac
      available = .true.
   end if
end function gasdev
```

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

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

```
subroutine verlet(coord, coord_t0, vel, vel_t0, acc, acc_t0, force, pener)
  use precision
  use potential
  use param, only : natom, mass, dt, boxl, pot
  implicit none
  real(dp), dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), intent(out) :: coord, vel, acc, force
  real(dp), intent(out) :: pener
  integer(ip) :: i, j, k
  real(dp) :: epot
  real(dp) :: r(3), f(3)

  ! Set coordinates, velocity, acceleration and force at next time step to zero
  coord = 0d0 ; vel = 0d0 ; acc = 0d0 ; force = 0d0
  pener = 0d0

  ! Get new atom positions from Velocity Verlet Algorithm
  coord = coord_t0 + vel_t0 * dt + 0.5d0 * acc_t0 * dt ** 2
  do i = 1, natom
     ! Apply PBC to coordinates
     where ( coord(i,:) > boxl(:) )
        coord(i,:) = coord(i,:) - boxl(:)
     elsewhere ( coord(i,:) < 0d0 )
        coord(i,:) = coord(i,:) + boxl(:)
     end where
  end do

  ! Get force at new atom positions
  do i = 1, natom - 1
     do j = i + 1, natom
        r(:) = coord(i,:) - coord(j,:)
        ! minimum image criterion
        r = r - nint( r / boxl ) * boxl
        select case(pot)
        case('mp')
           call morse( r, f, epot )
        case default
           call lennard_jones( r, f, epot )
        end select
        pener = pener + epot
        force(i,:) = force(i,:) + f(:)
        force(j,:) = force(j,:) - f(:)
     end do
  end do

  ! Calculate Acceleration and Velocity  at current time step
  acc = force / mass
  vel = vel_t0 + 0.5d0 * ( acc + acc_t0 ) * dt

end subroutine verlet
```

```
program md

. . .

  real(dp), dimension(:,:), allocatable :: coord_t0, coord
  real(dp), dimension(:,:), allocatable :: vel_t0, vel
  real(dp), dimension(:,:), allocatable :: acc_t0, acc, force
  real(dp) :: pener

  interface
     . . .
     subroutine verlet(coord, coord_t0, vel_t0, vel, acc_t0, acc, force, pener)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
       real(dp), dimension(:,:), intent(out) :: coord, vel, acc, force
       real(dp), intent(out) :: pener
     end subroutine verlet
     . . .
  end interface

  . . .

  do istep = 1, nstep

     ! Set coordinates, velocity, acceleration and force at next time step to
          zero
     coord = 0d0 ; vel = 0d0 ; acc = 0d0
     force = 0d0 ; pener = 0d0

     ! Get new atom positions from Velocity Verlet Algorithm
     call verlet(coord, coord_t0, vel_t0, vel, acc_t0, acc, force, pener)

     . . .
  end do

  ! Free up memory
  deallocate(coord_t0,vel_t0,acc_t0,coord,vel,acc,force)

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 involve the **call** statement

```
module potential
  use precision
  implicit none
  real(dp) :: r2, r6, d2, d
  real(dp), parameter :: de = 0.176d0, a = 1.4d0, re = 1d0
  real(dp) :: exparre

contains
  subroutine lennard_jones(r,f,p)
    ! Lennard Jones Potential
    ! V = 4 * epsilon * [ (sigma/r)**12 - (sigma/r)**6 ]
    !   = 4 * epsilon * (sigma/r)**6 * [ (sigma/r)**2 - 1 ]
    !   = 4 * r**(-6) * [ r**(-2) - 1 ] for epsilon=sigma=1
    ! F_i = 48 * epsilon * (sigma/r)**6 * (1/r**2) * [ ( sigma/r)** 6 - 0.5 ] *
              i where i = x,y,z
    !     = 48 * r**(-8) * [ r**(-6) - 0.5 ] * i  for epsilon=sigma=1
            implicit none
    implicit none
    real(dp), dimension(:), intent(in) :: r
    real(dp), dimension(:), intent(out) :: f
    real(dp), intent(out) :: p

    r2 = 1.d0 / dot_product(r,r)
    r6 = r2 ** 3

    f = dvdr_lj(r2, r6) * r
    p = pot_lj(r2, r6)
  end subroutine lennard_jones

  subroutine morse(r,f,p)
    ! Morse Potential
    ! V = D * [ 1 - exp(-a*(r - re)) ]^2
    ! F_i = 2*D * [ 1 - exp(-a*(r - re)) ] * a exp(-a*(r-re)) * i / r
    implicit none
    real(dp), dimension(:), intent(in) :: r
    real(dp), dimension(:), intent(out) :: f
    real(dp), intent(out) :: p

    d2 = dot_product(r,r)
    d = sqrt(d2)
    exparre = exp( -a * (d - re ))

    f = dvdr_mp(exparre) * r
    p = pot_mp(exparre)
  end subroutine morse

  function pot_lj(r2, r6)
    implicit none
    real(dp), intent(in) :: r2, r6
    real(dp) :: pot_lj
    pot_lj = 4d0 * r6 * ( r6 - 1.d0 )
  end function pot_lj
  function pot_mp(exparre)
    implicit none
    real(dp), intent(in) :: exparre
    real(dp) :: pot_mp
    pot_mp = de * ( 1d0 - exparre )**2
  end function pot_mp

  function dvdr_lj(r2,r6)
    implicit none
    real(dp), intent(in) :: r2, r6
    real(dp) :: dvdr_lj
    dvdr_lj = 48d0 * r2 * r6 * ( r6 - 0.5d0 )
  end function dvdr_lj
  function dvdr_mp(exparre)
    implicit none
    real(dp), intent(in) :: exparre
    real(dp) :: dvdr_mp
    dvdr_mp = 2d0 * de * a * (1d0 - exparre) * exparre
  end function dvdr_mp
end module potential
```

Array-valued Functions

- **`function`** can also return arrays

```
module potential
  use precision
  implicit none
  real(dp) :: r2, r6, d2, d
  real(dp), parameter :: de = 0.176d0, a = 1.4d0, re = 1d0
  real(dp) :: exparre

contains
  subroutine lennard_jones(r,f,p)
    ! Lennard Jones Potential
    ! V = 4 * epsilon * [ (sigma/r)**12 - (sigma/r)**6 ]
    !   = 4 * epsilon * (sigma/r)**6 * [ (sigma/r)**2 - 1 ]
    !   = 4 * r**(-6) * [ r**(-2) - 1 ] for epsilon=sigma=1
    ! F_i = 48 * epsilon * (sigma/r)**6 * (1/r**2) * [ ( sigma/r)** 6
             - 0.5 ] * i where i = x,y,z
    !     = 48 * r**(-8) * [ r**(-6) - 0.5 ] * i  for epsilon=sigma=1
                implicit none
    implicit none
    real(dp), dimension(:), intent(in) :: r
    real(dp), dimension(:), intent(out) :: f
    real(dp), intent(out) :: p

    r2 = 1.d0 / dot_product(r,r)
    r6 = r2 ** 3

    f = dvdr_lj(r2, r6, r)
    p = pot_lj(r2, r6)
  end subroutine lennard_jones

  subroutine morse(r,f,p)
    ! Morse Potential
    ! V = D * [ 1 - exp(-a*(r - re)) ]^2
    ! F_i = 2*D * [ 1 - exp(-a*(r - re)) ] * a exp(-a*(r-re)) * i / r
    implicit none
    real(dp), dimension(:), intent(in) :: r
    real(dp), dimension(:), intent(out) :: f
    real(dp), intent(out) :: p

    d2 = dot_product(r,r)
    d = sqrt(d2)
    exparre = exp( -a * (d - re ))

    f = dvdr_mp(exparre,r)
    p = pot_mp(exparre)
  end subroutine morse

  function pot_lj(r2, r6)
    implicit none
    real(dp), intent(in) :: r2, r6
    real(dp) :: pot_lj
    pot_lj = 4d0 * r6 * ( r6 - 1.d0 )
  end function pot_lj
  function pot_mp(exparre)
    implicit none
    real(dp), intent(in) :: exparre
    real(dp) :: pot_mp
    pot_mp = de * ( 1d0 - exparre )**2
  end function pot_mp

  function dvdr_lj(r2,r6,r)
    implicit none
    real(dp), intent(in) :: r2, r6, r
    real(dp), dimension(size(r)) :: dvdr_lj
    dvdr_lj = 48d0 * r2 * r6 * ( r6 - 0.5d0 ) * r
  end function dvdr_lj
  function dvdr_mp(exparre,r)
    implicit none
    real(dp), intent(in) :: exparre, r
    real(dp), dimension(size(r)) :: dvdr_mp
    dvdr_mp = 2d0 * de * a * (1d0 - exparre) * exparre * r
  end function dvdr_mp
end module potential
```

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

Argument Association

- Recall from MD code example the invocation

```
call linearmom(vel_t0)
```

- and the subroutine declaration

```
subroutine linearmom(vel)
```

- `vel_t0` is an actual argument and is associated with the dummy argument `vel`
- In `subroutine linearmom`, the name `vel` is an alias for `vel_t0`
- If the value of a dummy argument changes, then so does the value of the actual argument
- The actual and dummy arguments must correspond in type, kind and rank.

- In **subroutine** linearmom,

  i and vcm 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 linearmom(vel)
  use precision
  use param, only : natom
  implicit none
  real(dp), dimension(:,:), intent(inout) :: vel
  integer(ip) :: i
  real(dp) :: vcm(3)

  ! First get center of mass velocity
  vcm = 0d0
  do i = 1, 3
     vcm(i) = sum(vel(:,i))
  end do
  vcm = vcm / real(natom,dp)

  ! Now remove center of mass velocity from all atoms
  do i = 1, natom
     vel(i,:) = vel(i,:) - vcm(:)
  end do

end subroutine linearmom
```

Optional & Keyword Arguments I

- 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.

```
subroutine get_temp(vel,boltz)
  use precision
  use param, only : natom, avtemp, mass, kb
  implicit none
  real(dp), dimension(:,:), intent(in) :: vel
  real(dp), optional :: boltz
  integer(ip) :: i
  real(dp) :: ke

  if (present(boltz))kb = boltz
  ke = 0d0
  do i = 1, natom
     ke = ke + dot_product(vel(i,:),vel(i,:))
  end do
  avtemp = mass * ke / ( 3d0 * kb * real( natom - 1, dp))

end subroutine get_temp
```

```
subroutine initialize(coord_t0, vel_t0, acc_t0)
...
  interface
     subroutine linearmom(vel)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(inout) :: vel
     end subroutine linearmom
     subroutine get_temp(vel, boltz)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(in) :: vel
       real(dp), optional :: boltz
     end subroutine get_temp
  end interface
...
  call get_temp(vel_t0)
...
```

Optional & Keyword Arguments II

- 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 initialize(coord, vel, acc)

  ...
  real(dp),dimension(:,:), intent(out) :: coord, vel
        , acc
  ...
end subroutine initialize
```

```
program md
        ...
     call initialize(coord_t0, vel_t0, acc_t0)
        ...
end program md
```

Optional & Keyword Arguments III

- **subroutine** initialize can be invoked using
  1. using the positional argument invocation
  2. using keyword arguments

```
program md
...
  interface
     subroutine initialize(coord, vel, acc)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(out) :: coord, vel, acc
     end subroutine initialize
  end interface
...
! All three calls give the same result.
  call initialize(coord_t0, vel_t0, acc_t0)
  call initialize(coord=coord_t0, acc=acc_t0, vel=vel_t0)
  call initialize(coord_t0, acc=acc_t0, vel=vel_t0)
...
```

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
  3. *assumed-size*: final dimension is specified by $*$
     ```
     real :: assumed_size(dim1,dim2,*)
     ```
     Commomly used in FORTRAN, use assumed-shape arrays in Modern Fortran.
- dummy arguments cannot be (unallocated) allocatable arrays.

Explicit-shape Arrays

```
program md
  use precision
  use param
  implicit none
  integer(ip) :: n, i, j, k, l
  real(dp), dimension(:,:), allocatable :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), allocatable :: coord, vel, acc, force
  ...
  ! Allocate arrays
  allocate(coord(natom,3), coord_t0(natom,3))
  allocate(vel(natom,3), vel_t0(natom,3))
  allocate(acc(natom,3), acc_t0(natom,3))
  allocate(force(natom,3))

  !================================================
  ! Initialize coordinates and random velocities
  !================================================

  call initialize(coord_t0, vel_t0, acc_t0)

  ...
end program md

subroutine initialize(coord_t0, vel_t0, acc_t0)
  use precision
  use param, only : natom, npartdim, alat, rcell
  implicit none
  real(dp), dimension(natom,3) :: coord_t0, vel_t0, acc_t0
  integer(ip) :: n, i, j, k, l

  ! Set initial coordinates, velocity and acceleration to zero
  coord_t0 = 0d0 ; vel_t0 = 0d0 ; acc_t0 = 0d0
  ...
end subroutine initialize
```

Assumed-Shape Arrays

```
program md
  use precision
  use param
  implicit none
  integer(ip) :: n, i, j, k, l
  real(dp), dimension(:,:), allocatable :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), allocatable :: coord, vel, acc, force
  ...
  interface
     subroutine initialize(coord_t0, vel_t0, acc_t0)
       use precision
       implicit none
       real(dp), dimension(:,:), intent(out) :: coord_t0, vel_t0, acc_t0
     end subroutine initialize
     ...
  end interface
  ...
  ! Allocate arrays
  allocate(coord(natom,3), coord_t0(natom,3))
  allocate(vel(natom,3), vel_t0(natom,3))
  allocate(acc(natom,3), acc_t0(natom,3))
  allocate(force(natom,3))

  !================================================
  ! Initialize coordinates and random velocities
  !================================================

  call initialize(coord_t0, vel_t0, acc_t0)
  ...
end program md

subroutine initialize(coord_t0, vel_t0, acc_t0)
  use precision
  use param, only : natom, npartdim, alat, rcell
  implicit none
  real(dp), dimension(:,:), intent(out) :: coord_t0, vel_t0, acc_t0
  integer(ip) :: n, i, j, k, l

  ! Set initial coordinates, velocity and acceleration to zero
  coord_t0 = 0d0 ; vel_t0 = 0d0 ; acc_t0 = 0d0
  ...
end subroutine initialize
```

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,2)) :: 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 verlet(coord, coord_t0, vel_t0, vel, acc_t0, acc, force, pener)
  use precision
  use param, only : natom, mass, boxl, dt
  implicit none
  real(dp),dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
  real(dp),dimension(:,:), intent(out) :: coord, vel, acc, force
  real(dp), intent(out) :: pener

  .
  .
  .
end subroutine verlet
```

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

- 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(coord, coord_t0, vel, vel_t0, acc, acc_t0, force,
       pener)
  use precision
  use param, only : natom, mass, dt, boxl, pot
  implicit none
  real(dp), dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), intent(out) :: coord, vel, acc, force
  real(dp), intent(out) :: pener
  integer(ip) :: i

  interface
     subroutine get_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 get_pot_force
  end interface

  ! Set coordinates, velocity, acceleration and force at next time
       step to zero
  coord = 0d0 ; vel = 0d0 ; acc = 0d0

  ! Get new atom positions from Velocity Verlet Algorithm
  coord = coord_t0 + vel_t0 * dt + 0.5d0 * acc_t0 * dt ** 2
  do i = 1, natom
     ! Apply PBC to coordinates
     where ( coord(i,:) > boxl(:) )
        coord(i,:) = coord(i,:) - boxl(:)
     elsewhere ( coord(i,:) < 0d0 )
        coord(i,:) = coord(i,:) + boxl(:)
     end where
  end do

  ! Get Potential and force at new atom positions
  call get_pot_force(coord, force, pener)

  ! Calculate Acceleration and Velocity  at current time step
  acc = force / mass
  vel = vel_t0 + 0.5d0 * ( acc + acc_t0 ) * dt

end subroutine verlet

subroutine get_pot_force(coord, force, pener)
  use precision
  use potential
  use param, only : natom, boxl
  implicit none
  real(dp), dimension(:,:), intent(in) :: coord
  real(dp), dimension(:,:), intent(out) :: force
  real(dp), intent(out) :: pener
  integer(ip) :: i, j
  real(dp) :: epot
  real(dp) :: r(3), f(3)

  pener = 0d0
  force = 0d0
  do i = 1, natom - 1
     do j = i + 1, natom
        r(:) = coord(i,:) - coord(j,:)
        ! minimum image criterion
        r = r - nint( r / boxl ) * boxl
        select case(pot)
        case('mp')
           call morse( r, f, epot )
        case default
           call lennard_jones( r, f, epot )
        end select
        pener = pener + epot
        force(i,:) = force(i,:) + f(:)
        force(j,:) = force(j,:) - f(:)
     end do
  end do

end subroutine get_pot_force
```

Interfaces III

```
subroutine verlet(coord, coord_t0, vel, vel_t0, acc, acc_t0, force,
       pener)
  use precision
  use param, only : natom, mass, dt, boxl, pot
  implicit none
  real(dp), dimension(:,:), intent(in) :: coord_t0, vel_t0, acc_t0
  real(dp), dimension(:,:), intent(out) :: coord, vel, acc, force
  real(dp), intent(out) :: pener
  integer(ip) :: i

  ! Set coordinates, velocity, acceleration and force at next time
         step to zero
  coord = 0d0 ; vel = 0d0 ; acc = 0d0

  ! Get new atom positions from Velocity Verlet Algorithm
  coord = coord_t0 + vel_t0 * dt + 0.5d0 * acc_t0 * dt ** 2
  do i = 1, natom
     ! Apply PBC to coordinates
     where ( coord(i,:) > boxl(:) )
        coord(i,:) = coord(i,:) - boxl(:)
     elsewhere ( coord(i,:) < 0d0 )
        coord(i,:) = coord(i,:) + boxl(:)
     end where
  end do

  ! Get Potential and force at new atom positions
  call get_pot_force(coord, force, pener)

  ! Calculate Acceleration and Velocity  at current time step
  acc = force / mass
  vel = vel_t0 + 0.5d0 * ( acc + acc_t0 ) * dt

contains
```

```
  subroutine get_pot_force(coord, force, pener)
    use potential
    implicit none
    real(dp), dimension(:,:), intent(in) :: coord
    real(dp), dimension(:,:), intent(out) :: force
    real(dp), intent(out) :: pener
    integer(ip) :: i, j
    real(dp) :: epot
    real(dp) :: r(3), f(3)

    pener = 0d0
    force = 0d0
    do i = 1, natom - 1
       do j = i + 1, natom
          r(:) = coord(i,:) - coord(j,:)
          ! minimum image criterion
          r = r - nint( r / boxl ) * boxl
          select case(pot)
          case('mp')
             call morse( r, f, epot )
          case default
             call lennard_jones( r, f, epot )
          end select
          pener = pener + epot
          force(i,:) = force(i,:) + f(:)
          force(j,:) = force(j,:) - f(:)
       end do
    end do

  end subroutine get_pot_force

end subroutine verlet
```

- Here since **subroutine** get_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
  save
  integer, parameter :: ip = selected_int_kind(15)
  integer, parameter :: dp = selected_real_kind(15)
end module precision

module param
  use precision
  implicit none
  integer(ip) :: npartdim, natom, nstep, istep
  real(dp) :: tempK, dt, boxl(3), alat, mass
  real(dp) :: avtemp, ke, kb, epsilon, sigma, scale
  real(dp),dimension(3,4) :: rcell = reshape( (/ &
       0.0D+00, 0.0D+00, 0.0D+00, &
       0.5D+00, 0.5D+00, 0.0D+00, &
       0.0D+00, 0.5D+00, 0.5D+00, &
       0.5D+00, 0.0D+00, 0.5D+00 /), (/ 3, 4 /) )
  character(len=2) :: pot
end module param
```

- 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.

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 potential`, all functions which calculate forces can be declared as private as follows

```
module potential
  use precision
  implicit none
  real(dp) :: r2, r6, d2, d
  real(dp), parameter :: de = 0.176d0, a = 1.4d0, re = 1d0
  real(dp) :: exparre
  public :: lennard_jones, morse, pot_lj, pot_mp
  private :: dvdr_lj, dvdr_mp

contains
  ...
end module potential
```

- Program Units in the MD code can directly call `lennard_jones`, `morse`, `pot_lj` and `pot_mp` but cannot access `dvdr_lj` and `dvdr_mp`

Using Modules

- 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 dt to more user readable variables:

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

- It's good programming practice to use only those variables from modules that are neccessary to avoid name conflicts and overwrite variables.
- For this, use the **use** <modulename>, **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 `potential.f90` and subroutines `initialize.f90`, `verlet.f90`, `linearmom.f90` and `get_temp.f90`.
- In general, the code can be compiled as

```
ifort -o md md.f90 precision.f90 param.f90 potential.f90 initialize.f90 \
   verlet.f90 linearmom.f90 get_temp.f90
```

- Most compilers are restrictive in the order of compilation.
- The order in which the sub programs should be compiled is
  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 `potential` 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 md.f90 precision.f90 param.f90 potential.f90 initialize.f90 \
   verlet.f90 linearmom.f90 get_temp.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 potential.f90
        creates precision.mod param.mod potential.mod
```

- The main program can now be compiled as

```
ifort -o md md.f90 initialize.f90 verlet.f90 linearmom.f90 get_temp.f90 \
   -I{path to directory containing the .mod files}
```

- The Makefile tutorial will cover this aspect in more detail.

Derived Types

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 operator 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 ! defines x1,y1,x2,y2
  integer :: direction ! 0=nodirection, 1=(x1,y1)->(x2,y2)
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)
```

Derived Types III

- 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.

```
coord_t0(n)%x = alat * real(i - 1, dp) + rcell(1,1)
coord_t0(n)%y = alat * real(j - 1, dp) + rcell(2,1)
coord_t0(n)%z = alat * real(k - 1, dp) + rcell(3,1)
  OR
x = alat * real(i - 1, dp) + rcell(1,1)
y = alat * real(j - 1, dp) + rcell(2,1)
z = alat * real(k - 1, dp) + rcell(3,1)
coord_t0(n) = dynamics( x, y, z )
```

- I/O on Derived Types
  - Can do normal I/O on derived types
    `print *, a` will produce the result `1.00.51.5`
    `print *, c` will produce the result `2.00.00.00.0`
- Arrays and Derived Types
  - Can define derived type objects which contain non-allocatable arrays and arrays of derived type objects
- Derived Type Valued Functions

Derived Types IV

- Functions can return results of an arbitrary defined type.

- 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, private :: 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`

- 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`

Generic 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,coord
       0,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.

- 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 operations are now defined for derived types `a,b,c` and scalar `r`

```
c = a * r
c = r * a
c = a + b
```

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

```
c%x = a%x * r
c%y = a%y * r
c%z = a%z * r

c%x = r * a%x
c%y = r * a%y
c%z = r * a%z

c%x = a%x + b%x
c%y = a%y + b%y
c%z = a%z + b%z
```

Object Based Programming

- 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 (covered)
  5. polymorphism - user can program his/her own polymorphism by generic overloading
  6. resuability - modules

- 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

- **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

- **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`

- 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)
```

- 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

- 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)
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
```

- 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 referenced as `c%x1, c%y1, c%x1, c%y2` and `c%direction`
- Similarly, an object, `d` of type `painted_line_type` is referenced 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`

- 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 coverage 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`
- Scientific Programming in Fortran 2003: A tutorial Including Object-Oriented Programming, Katherine Holcomb, University of Virginia.

Exercise

- Molecular Dynamics code for melting of solid Argon using Lennard-Jones Potential.
- Your goal is to rewrite the code using Modern Fortran concepts that you have grasped.
- This exercise is more of a "What concepts have I learned of Modern Fortran?", so there are multiple correct solutions
- Code can be obtained from `http://www.hpc.lsu.edu/training/archive/tutorials.php`:
- md-orig.f90 is the original code that you should begin working on (this is the same code that was shown in todays slides)
- There is no "correct solution", however there are multiple solutions md-v{1-5}.f90 based on various concepts presented.
- It's entirely up to you to decide which solution you want to arrive at.
- Compare the results of your edited code with that of md-v0.out. If the results are not the same, debug your code.

Calculate pi by Numerical Integration I

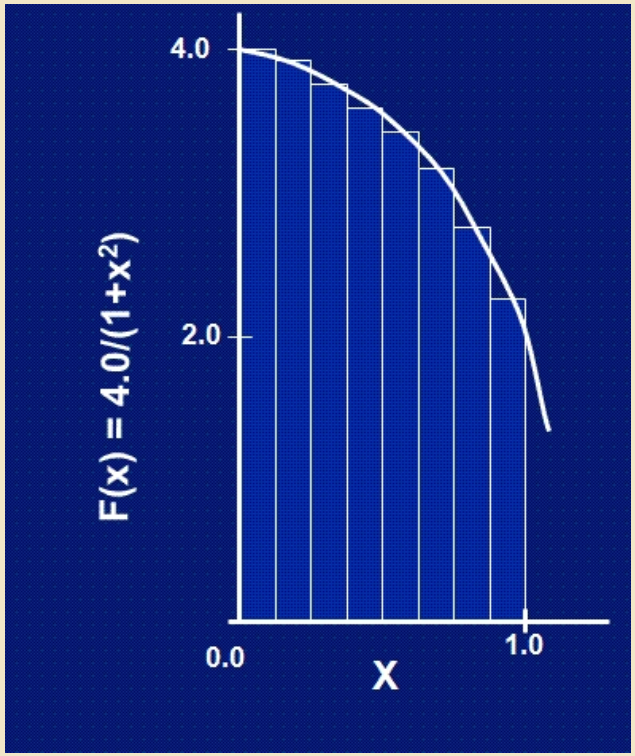
- We know that

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

- So numerically, we can approxiate pi as the sum of a number of rectangles

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Meadows et al, A “hands-on” introduction to OpenMP, SC09



Calculate pi by Numerical Integration II

Algorithm 4 Pseudo Code for Calculating Pi

program CALCULATE_PI
  $step \leftarrow 1/n$
  $sum \leftarrow 0$
  do $i \leftarrow 0 \cdots n$
    $x \leftarrow (i + 0.5) * step; sum \leftarrow sum + 4/(1 + x^2)$
  end do
  $pi \leftarrow sum * step$
end program

- SAXPY is a common operation in computations with vector processors included as part of the BLAS routines

  $y \leftarrow \alpha x + y$

- Write a SAXPY code to multiply a vector with a scalar.

Algorithm 5 Pseudo Code for SAXPY

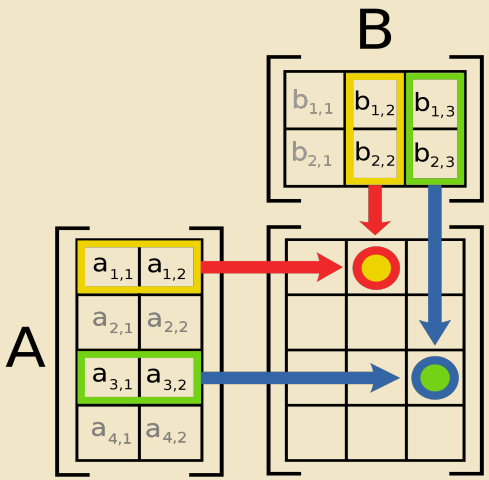
---

program SAXPY
  $n \leftarrow$ some large number
  $x(1:n) \leftarrow$ some number say, 1
  $y(1:n) \leftarrow$ some other number say, 2
  $a \leftarrow$ some other number ,say, 3
  do $i \leftarrow 1 \cdots n$
    $y_i \leftarrow y_i + a * x_i$
  end do
end program SAXPY

Matrix Multiplication I

- Most Computational code involve matrix operations such as matrix multiplication.
- Consider a matrix **C** which is a product of two matrices **A** and **B**:

  Element *i,j* of **C** is the dot product of the i^{th} row of **A** and j^{th} column of **B**
- Write a MATMUL code to multiple two matrices.



Matrix Multiplication II

Algorithm 6 Pseudo Code for MATMUL

program MATMUL
  $m, n \leftarrow$ some large number ≤ 1000
  Define a_{mn}, b_{nm}, c_{mm}
  $a_{ij} \leftarrow i + j; b_{ij} \leftarrow i - j; c_{ij} \leftarrow 0$
  do $i \leftarrow 1 \cdots m$
    do $j \leftarrow 1 \cdots m$
      $c_{i,j} \leftarrow \sum_{k=1}^{n} a_{i,k} * b_{k,j}$
    end do
  end do
end program MATMUL