LEHIGH UNIVERSITY

Modern Fortran Programming I

Alexander B. Pacheco
LTS Research Computing
April 28, 2015

Outline

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

```
C2345678901234567890123456789012345678901234567890123456789012345678901234567890
      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.

$$\underbrace{ABCDEFGH}_{REAL}\overbrace{IJKLMN}^{INTEGER}\underbrace{OPQRSTUVWXYZ}_{REAL}$$

- 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

```fortran
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
	tn	tab to position n
Tabs	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
```

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

```
[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

```fortran
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

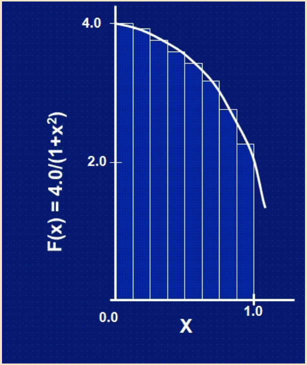
- We know that

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

- 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.

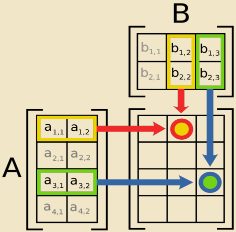
Algorithm 2 Pseudo Code for SAXPY

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