Fortran 90 Control Structures

Computer programming is an art form,
like the creation of poetry or music.

Donald Ervin Knuth

LOGICAL Variables

- A **LOGIAL** variable can only hold either **.TRUE.** or **.FALSE.** , and cannot hold values of any other type.
- Use **T** or **F** for **LOGICAL** variable **READ(*,*)**
- **WRITE(*,*)** prints **T** or **F** for **.TRUE.** and **.FALSE.**, respectively.

```
LOGICAL, PARAMETER :: Test = .TRUE.
LOGICAL            :: C1, C2

C1 = .true.      ! correct
C2 = 123         ! Wrong
READ(*,*)  C1, C2
C2 = .false.
WRITE(*,*) C1, C2
```

Relational Operators: 1/4

- Fortran 90 has six relational operators: `<`, `<=`, `>`, `>=`, `==`, `/=`.
- Each of these six relational operators takes two expressions, compares their values, and yields `.TRUE.` or `.FALSE.`
- Thus, `a < b < c` is wrong, because `a < b` is `LOGICAL` and `c` is `REAL` or `INTEGER`.
- `COMPLEX` values can only use `==` and `/=`
- `LOGICAL` values should use `.EQV.` or `.NEQV.` for equal and not-equal comparison.

Relational Operators: 2/4

- **Relational operators have *lower* priority than arithmetic operators, and `//`.**
- **Thus, `3 + 5 > 10` is `.FALSE.` and `"a" // "b" == "ab"` is `.TRUE.`**
- **Character values are encoded. Different standards (*e.g.*, BCD, EBCDIC, ANSI) have different encoding sequences.**
- **These encoding sequences may not be compatible with each other.**

Relational Operators: 3/4

- **For maximum portability, only assume the following orders for letters and digits.**
- **Thus, `"A" < "X"`, `'f' <= "u"`, and `"2" < "7"` yield `.TRUE.` But, we don't know the results of `"S" < "s"` and `"t" >= "%"`.**
- **However, equal and not-equal such as `"S" /= "s"` and `"t" == "5"` are fine.**

```
A < B < C < D < E < F < G < H < I < J < K < L < M < N
          < O < P < Q < R < S < T < U < V < W < X < Y < Z

a < b < c < d < e < f < g < h < i < j < k < l < m < n
          < o < p < q < r < s < t < u < v < w < x < y < z

0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9
```

Relational Operators: 4/4

- **String comparison rules:**
  - **Start scanning from the first character.**
  - **If the current two are equal, go for the next**
    - **If there is no more characters to compare, the strings are equal (*e.g.*, `"abc" == "abc"`)**
    - **If one string has no more character, the shorter string is smaller (*e.g.*, `"ab" < "abc"` is `.TRUE.`)**
  - **If the current two are not equal, the string has the smaller character is smaller (*e.g.*, `"abcd"` is smaller than `"abct"`).**

LOGICAL Operators: 1/2

- **There are 5 `LOGICAL` operators in Fortran 90: `.NOT.`, `.OR.`, `.AND.`, `.EQV.` and `.NEQV.`**
- **`.NOT.` is the highest, followed by `.OR.` and `.AND.`, `.EQV.` and `.NEQV.` are the lowest.**
- **Recall that `.NOT.` is evaluated from *right to left*.**
- **If both operands of `.EQV.` (*equivalence*) are the same, `.EQV.` yields `.TRUE.`.**
- **`.NEQV.` is the opposite of `.EQV.` (*not equivalence*). If the operands of `.NEQV.` have different values, `.NEQV.` yields `.TRUE.`**

LOGICAL Operators: 2/2

- **If INTEGER variables m, n, x and y have values 3, 5, 4 and 2, respectively.**

```
.NOT. (m > n .AND. x < y) .NEQV. (m <= n .AND. x >= y)
   → .NOT. (3 > 5 .AND. 4 < 2) .NEQV. (3 <= 5 .AND. 4 >= 2)
   → .NOT. (.FALSE. .AND. 4 < 2) .NEQV. (3 <= 5 .AND. 4 >= 2)
   → .NOT. (.FALSE. .AND. .FALSE.) .NEQV. (3 <= 5 .AND. 4 >= 2)
   → .NOT. .FALSE. .NEQV. (3 <= 5 .AND. 4 >= 2)
   → .TRUE. .NEQV. (3 <= 5 .AND. 4 >= 2)
   → .TRUE. .NEQV. (.TRUE. .AND. 4 >= 2)
   → .TRUE. .NEQV. (.TRUE. .AND. .TRUE.)
   → .TRUE. .NEQV. .TRUE.
   → .FALSE.
```

.NOT. is higher than .NEQV.

`IF-THEN-ELSE` Statement: 1/4

- Fortran 90 has three if-then-else forms.
- The most complete one is the `IF-THEN-ELSE-IF-END IF`
- An old logical `IF` statement may be very handy when it is needed.
- There is an old and obsolete arithmetic `IF` that you are not encouraged to use. We won't talk about it at all.
- Details are in the next few slides.

`IF-THEN-ELSE` Statement: 2/4

- **`IF-THEN-ELSE-IF-END IF` is the following.**
- **Logical expressions are evaluated sequentially (*i.e.*, top-down). The statement sequence that corresponds to the expression evaluated to `.TRUE.` will be executed.**
- **Otherwise, the `ELSE` sequence is executed.**

```
IF (logical-expression-1) THEN
    statement sequence 1
ELSE IF (logical-expression-2) THEN
    statement seqence 2
ELSE IF (logical-expression-3) THEN
    statement sequence 3
ELSE IF (.....) THEN
    ...........
ELSE
    statement sequence ELSE
END IF
```

IF-THEN-ELSE Statement: 3/4

- **Two Examples:**

Find the minimum of **a**, **b** *and* **c** *and saves the result to* **Result**

```
IF (a < b .AND. a < c) THEN
   Result = a
ELSE IF (b < a .AND. b < c) THEN
   Result = b
ELSE
   Result = c
END IF
```

Letter grade for **x**

```
INTEGER :: x
CHARACTER(LEN=1) :: Grade

IF (x < 50) THEN
   Grade = 'F'
ELSE IF (x < 60) THEN
   Grade = 'D'
ELSE IF (x < 70) THEN
   Grade = 'C'
ELSE IF (x < 80) THEN
   Grade = 'B'
ELSE
   Grade = 'A'
END IF
```

IF-THEN-ELSE Statement: 4/4

- The ELSE-IF part and ELSE part are optional.
- If the ELSE part is missing and none of the logical expressions is .TRUE., the IF-THEN-ELSE has no effect.

no ELSE-IF

```
IF (logical-expression-1) THEN
     statement sequence 1
ELSE
     statement sequence ELSE
END IF
```

no ELSE

```
IF (logical-expression-1) THEN
     statement sequence 1
ELSE IF (logical-expression-2) THEN
     statement sequence 2
ELSE IF (logical-expression-3) THEN
     statement sequence 3
ELSE IF (.....) THEN
     ...........
END IF
```

Example: 1/2

- **Given a quadratic equation $ax^2 + bx + c = 0$, where $a \neq 0$, its roots are computed as follows:**

$$x = \frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$$

- **However, this is a very poor and unreliable way of computing roots. Will return to this soon.**

```
PROGRAM QuadraticEquation
   IMPLICIT NONE
   REAL :: a, b, c
   REAL :: d
   REAL :: root1, root2
   …… other executable statement ……
END PROGRAM QuadraticEquation
```

Example: 2/2

- **The following shows the executable part**

```
READ(*,*)  a, b, c
WRITE(*,*) 'a = ', a
WRITE(*,*) 'b = ', b
WRITE(*,*) 'c = ', c
WRITE(*,*)

d = b*b - 4.0*a*c
IF (d >= 0.0) THEN                            ! is it solvable?
   d = SQRT(d)
   root1 = (-b + d)/(2.0*a)                   ! first root
   root2 = (-b - d)/(2.0*a)                   ! second root
   WRITE(*,*) 'Roots are ', root1, ' and ', root2
ELSE                                          ! complex roots
   WRITE(*,*) 'There is no real roots!'
   WRITE(*,*) 'Discriminant = ', d
END IF
```

IF-THEN-ELSE Can be Nested: 1/2

- Another look at the quadratic equation solver.

```
IF (a == 0.0) THEN                        ! could be a linear equation
   IF (b == 0.0) THEN                     ! the input becomes c = 0
      IF (c == 0.0) THEN                  ! all numbers are roots
         WRITE(*,*) 'All numbers are roots'
      ELSE                                ! unsolvable
         WRITE(*,*) 'Unsolvable equation'
      END IF
   ELSE                                   ! linear equation bx + c = 0
      WRITE(*,*) 'This is linear equation, root = ', -c/b
   END IF
ELSE                                ! ok, we have a quadratic equation
   ...... solve the equation here ……
END IF
```

IF-THEN-ELSE Can be Nested: 2/2

- Here is the big **ELSE** part:

```
d = b*b - 4.0*a*c
IF (d > 0.0) THEN                         ! distinct roots?
   d = SQRT(d)
   root1 = (-b + d)/(2.0*a)               ! first root
   root2 = (-b - d)/(2.0*a)               ! second root
   WRITE(*,*) 'Roots are ', root1, ' and ', root2
ELSE IF (d == 0.0) THEN                   ! repeated roots?
   WRITE(*,*) 'The repeated root is ', -b/(2.0*a)
ELSE                                      ! complex roots
   WRITE(*,*) 'There is no real roots!'
   WRITE(*,*) 'Discriminant = ', d
END IF
```

Logical **IF**

- **The logical `IF` is from Fortran 66, which is an improvement over the Fortran I arithmetic `IF`.**
- **If logical-expression is `.TRUE.` , *statement* is executed. Otherwise, execution goes though.**
- **The statement can be assignment and input/output.**

```
IF (logical-expression) statement
```

```
Smallest = b
IF (a < b)  Smallest = a
```

```
Cnt = Cnt + 1
IF (MOD(Cnt,10) == 0) WRITE(*,*) Cnt
```

The SELECT CASE Statement: 1/7

- **Fortran 90 has the `SELECT CASE` statement for selective execution if the selection criteria are based on simple values in `INTEGER`, `LOGICAL` and `CHARACTER`. No, `REAL` is not applicable.**

```
SELECT CASE (selector)
   CASE (label-list-1)
      statements-1
   CASE (label-list-2)
      statements-2
   CASE (label-list-3)
      statements-3
   ...... other cases ......
   CASE (label-list-n)
      statements-n
   CASE DEFAULT
      statements-DEFAULT
END SELECT
```

selector **is an expression evaluated to an `INTEGER`, `LOGICAL` or `CHARACTER` value**

label-list **is a set of constants or `PARAMETERS` of the same type as the *selector***

statements **is one or more executable statements**

The `SELECT CASE` Statement: 2/7

- **The *label-list* is a list of the following forms:**
  - **`value` → a specific value**
  - **`value1 : value2` → values between `value1` and `value2`, including `value1` and `value2`, and `value1 <= value2`**
  - **`value1 :` → values larger than or equal to `value1`**
  - **`: value2` → values less than or equal to `value2`**
- **Reminder: `value`, `value1` and `value2` must be constants or `PARAMETER`s.**

The SELECT CASE Statement: 3/7

- The `SELECT CASE` statement is executed as follows:
  - Compare the value of *selector* with the labels in each case. If a match is found, execute the corresponding *statements*.
  - If no match is found and if `CASE DEFAULT` is there, execute the *statements-*`DEFAULT`.
  - Execute the next statement following the `SELECT CASE`.

```
SELECT CASE (selector)
   CASE (label-list-1)
      statements-1
   CASE (label-list-2)
      statements-2
   CASE (label-list-3)
      statements-3
   ...... other cases ......
   CASE (label-list-n)
      statements-n
   CASE DEFAULT
      statements-DEFAULT
END SELECT
```

optional

The `SELECT CASE` Statement: 4/7

- **Some important notes:**
  - **The values in *label-list*s should be unique. Otherwise, it is not known which `CASE` would be selected.**
  - **`CASE DEFAULT` should be used whenever it is possible, because it guarantees that there is a place to do something (*e.g.*, error message) if no match is found.**
  - **`CASE DEFAULT` can be anywhere in a `SELECT CASE` statement; but, a preferred place is the last in the `CASE` list.**

The SELECT CASE Statement: 5/7

- Two examples of **SELECT CASE**:

```
CHARACTER(LEN=4) :: Title
INTEGER :: DrMD = 0, PhD = 0
INTEGER :: MS = 0, BS = 0
INTEGER ::Others = 0

SELECT CASE (Title)
  CASE ("DrMD")
    DrMD = DrMD + 1
  CASE ("PhD")
    PhD = PhD + 1
  CASE ("MS")
    MS = MS + 1
  CASE ("BS")
    BS = BS + 1
  CASE DEFAULT
    Others = Others + 1
END SELECT
```

```
CHARACTER(LEN=1) :: c

SELECT CASE (c)
  CASE ('a' : 'j')
    WRITE(*,*) 'First ten letters'
  CASE ('l' : 'p', 'u' : 'y')
    WRITE(*,*)                          &
        'One of l,m,n,o,p,u,v,w,x,y'
  CASE ('z', 'q' : 't')
    WRITE(*,*) 'One of z,q,r,s,t'
  CASE DEFAULT
    WRITE(*,*) 'Other characters'
END SELECT
```

The SELECT CASE Statement: 6/7

- Here is a more complex example:

```
INTEGER :: Number, Range

SELECT CASE (Number)
   CASE ( : -10, 10 : )
      Range = 1
   CASE (-5:-3, 6:9)
      Range = 2
   CASE (-2:2)
      Range = 3
   CASE (3, 5)
      Range = 4
   CASE (4)
      Range = 5
   CASE DEFAULT
      Range = 6
END SELECT
```

Number	*Range*	*Why?*
`<= -10`	1	`CASE (:-10, 10:)`
`-9,-8,-7,-6`	6	`CASE DEFAULT`
`-5,-4,-3`	2	`CASE (-5:-3, 6:9)`
`-2,-1,0,1,2`	3	`CASE (-2:2)`
`3`	4	`CASE (3, 5)`
`4`	5	`CASE (4)`
`5`	4	`CASE (3, 5)`
`6,7,8,9`	2	`CASE (-5:-3, 6:9)`
`>= 10`	1	`CASE (:-10, 10:)`

The SELECT CASE Statement: 7/7

This program reads in a character and determines if it is a vowel, a consonant, a digit, one of the four arithmetic operators, a space, or something else (*i.e.*, %, $, @, etc).

```
PROGRAM CharacterTesting
  IMPLICIT NONE
  CHARACTER(LEN=1) :: Input
  READ(*,*) Input
  SELECT CASE (Input)
  CASE ('A' : 'Z', 'a' : 'z')                   ! rule out letters
    WRITE(*,*) 'A letter is found : "', Input, '"'
    SELECT CASE (Input)                         ! a vowel ?
      CASE ('A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o','u')
        WRITE(*,*) 'It is a vowel'
      CASE DEFAULT                              ! it must be a consonant
        WRITE(*,*) 'It is a consonant'
    END SELECT
  CASE ('0' : '9')                              ! a digit
    WRITE(*,*) 'A digit is found : "', Input, '"'
  CASE ('+', '-', '*', '/')                     ! an operator
    WRITE(*,*) 'An operator is found : "', Input, '"'
  CASE (' ')                                    ! space
    WRITE(*,*) 'A space is found : "', Input, '"'
  CASE DEFAULT                                  ! something else
    WRITE(*,*) 'Something else found : "', Input, '"'
  END SELECT
END PROGRAM CharacterTesting
```

The Counting **DO** Loop: 1/6

- **Fortran 90 has two forms of `DO` loop: the counting `DO` and the general `DO`.**
- **The counting `DO` has the following form:**

```
DO control-var = initial, final [, step]
   statements
END DO
```

- **`control-var` is an `INTEGER` variable, `initial`, `final` and `step` are `INTEGER` expressions; however, `step` *cannot be zero*.**
- **If `step` is omitted, its default value is `1`.**
- ***statements* are executable statements of the `DO`.**

The Counting DO Loop: 2/6

- **Before a `DO`-loop starts, expressions `initial`, `final` and `step` are evaluated *exactly once*. When executing the `DO`-loop, these values will *not* be re-evaluated.**
- **Note again, the value of `step` *cannot be zero*.**
- **If `step` is positive, this `DO` counts up; if `step` is negative, this `DO` counts down**

```
DO control-var = initial, final [, step]
     statements
END DO
```

The Counting `DO` Loop: 3/6

- **If `step` is positive:**
  - **The `control-var` receives the value of `initial`.**
  - **If the value of `control-var` is less than or equal to the value of `final`, the *statements* part is executed. Then, the value of `step` is added to `control-var`, and goes back and compares the values of `control-var` and `final`.**
  - **If the value of `control-var` is greater than the value of `final`, the `DO`-loop completes and the statement following `END DO` is executed.**

The Counting DO Loop: 4/6

- **If `step` is negative:**
  - **The `control-var` receives the value of `initial`.**
  - **If the value of `control-var` is greater than or equal to the value of `final`, the *statements* part is executed. Then, the value of `step` is added to `control-var`, goes back and compares the values of `control-var` and `final`.**
  - **If the value of `control-var` is less than the value of `final`, the `DO`-loop completes and the statement following `END DO` is executed.**

The Counting DO Loop: 5/6

- **Two simple examples:**

```
INTEGER :: N, k

READ(*,*)  N
WRITE(*,*) "Odd number between 1 and ", N
DO k = 1, N, 2
   WRITE(*,*) k
END DO
```

odd integers between 1 & N

```
INTEGER, PARAMETER :: LONG = SELECTED_INT_KIND(15)
INTEGER(KIND=LONG) :: Factorial, i, N

READ(*,*)  N
Factorial = 1_LONG
DO i = 1, N
   Factorial = Factorial * i
END DO
WRITE(*,*) N, "! = ", Factorial
```

factorial of N

The Counting `DO` Loop: 6/6

- **Important Notes:**
  - **The step size `step`** ***cannot be zero***
  - **Never change the value of any variable in `control-var` and `initial`, `final`, and `step`.**
  - **For a count-down `DO`-loop, `step` must be negative. Thus, "`do i = 10, -10`" is not a count-down `DO`-loop, and the** ***statements*** **portion is not executed.**
  - **Fortran 77 allows `REAL` variables in `DO`; but, don't use it as it is not safe.**

General **DO**-Loop with **EXIT**: 1/2

- **The general DO-loop has the following form:**

```
DO
      statements
END DO
```

- ***statements*** **will be executed repeatedly.**
- **To exit the `DO`-loop, use the `EXIT` or `CYCLE` statement.**
- **The `EXIT` statement brings the flow of control to the statement following (*i.e.*, exiting) the `END DO`.**
- **The `CYCLE` statement starts the next iteration (*i.e.*, executing *statements* again).**

General DO-Loop with EXIT: 2/2

```
REAL, PARAMETER :: Lower = -1.0, Upper = 1.0, Step = 0.25
REAL :: x

x = Lower                   ! initialize the control variable
DO
   IF (x > Upper) EXIT      ! is it > final-value?
   WRITE(*,*) x             ! no, do the loop body
   x = x + Step             ! increase by step-size
END DO
```

```
INTEGER :: Input

DO
   WRITE(*,*) 'Type in an integer in [0, 10] please --> '
   READ(*,*) Input
   IF (0 <= Input .AND. Input <= 10) EXIT
   WRITE(*,*) 'Your input is out of range. Try again'
END DO
```

Example, **exp(*x*)**: 1/2

- **The exp(*x*) function has an infinite series:**

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^i}{i!} + \ldots\ldots$$

- **Sum each term until a term's absolute value is less than a tolerance, say 0.00001.**

```
PROGRAM Exponential
   IMPLICIT NONE
   INTEGER :: Count                             ! # of terms used
   REAL :: Term                                 ! a term
   REAL :: Sum                                  ! the sum
   REAL :: X                                    ! the input x
   REAL, PARAMETER :: Tolerance = 0.00001 ! tolerance
   …… executable statements ……
END PROGRAM Exponential
```

Example, exp(x): 2/2

- **Note:** $$\frac{x^{i+1}}{(i+1)!} = \left(\frac{x^i}{i!}\right) \times \left(\frac{x}{i+1}\right)$$
- **This is *not* a good solution, though.**

```
READ(*,*) X                          ! read in x
Count = 1                            ! the first term is 1
Sum   = 1.0                          ! thus, the sum starts with 1
Term  = X                            ! the second term is x
DO                                   ! for each term
   IF (ABS(Term) < Tolerance) EXIT         ! if too small, exit
   Sum = Sum + Term                  ! otherwise, add to sum
   Count = Count + 1                 ! count indicates the next term
   Term = Term * (X / Count)         ! compute the value of next term
END DO
WRITE(*,*) 'After ', Count, ' iterations:'
WRITE(*,*) ' Exp(', X, ') = ', Sum
WRITE(*,*) ' From EXP()   = ', EXP(X)
WRITE(*,*) ' Abs(Error)   = ', ABS(Sum - EXP(X))
```

Example, Prime Checking: 1/2

- **A positive integer *n* >= 2 is a *prime* number if the only divisors of this integer are 1 and itself.**
- **If *n* = 2, it is a prime.**
- **If *n* > 2 is even (*i.e.*, `MOD`(*n*,2) == 0), not a prime.**
- **If *n* is odd, then:**
  - **If the odd numbers between 3 and *n*-1 cannot divide *n*, *n* is a prime!**
  - **Do we have to go up to *n*-1? No, `SQRT`(*n*) is good enough. Why?**

Example, Prime Checking: 2/2

```
INTEGER :: Number                                   ! the input number
INTEGER :: Divisor                                  ! the running divisor

READ(*,*) Number                                    ! read in the input
IF (Number < 2) THEN                                ! not a prime if < 2
   WRITE(*,*) 'Illegal input'
ELSE IF (Number == 2) THEN                          ! is a prime if = 2
   WRITE(*,*) Number, ' is a prime'
ELSE IF (MOD(Number,2) == 0) THEN                   ! not a prime if even
   WRITE(*,*) Number, ' is NOT a prime'
ELSE                                                ! an odd number here
   Divisor = 3                                      ! divisor starts with 3
   DO                                               ! divide the input number
      IF (Divisor*Divisor > Number .OR. MOD(Number, Divisor) == 0) EXIT
      Divisor = Divisor + 2                         ! increase to next odd
   END DO
   IF (Divisor*Divisor > Number) THEN               ! which condition fails?
      WRITE(*,*) Number, ' is a prime'
   ELSE
      WRITE(*,*) Number, ' is NOT a prime'
   END IF
END IF
```

this is better than `SQRT(REAL(Divisor)) > Number`

Finding All Primes in [2,*n*]: 1/2

- **The previous program can be modified to find all prime numbers between 2 and *n*.**

```
PROGRAM Primes
  IMPLICIT NONE
  INTEGER :: Range, Number, Divisor, Count

  WRITE(*,*) 'What is the range ? '
  DO                              ! keep trying to read a good input
    READ(*,*) Range               ! ask for an input integer
    IF (Range >= 2) EXIT          ! if it is GOOD, exit
    WRITE(*,*) 'The range value must be >= 2. Your input = ', Range
    WRITE(*,*) 'Please try again:' ! otherwise, bug the user
  END DO
  …… we have a valid input to work on here ……
END PROGRAM Primes
```

Finding All Primes in [2,*n*]: 2/2

```
Count = 1                                    ! input is correct. start counting
WRITE(*,*)                                   ! 2 is a prime
WRITE(*,*) 'Prime number #', Count, ': ', 2

DO Number = 3, Range, 2                      ! try all odd numbers 3, 5, 7, ...
  Divisor = 3                                ! divisor starts with 3
  DO
    IF (Divisor*Divisor > Number .OR. MOD(Number,Divisor) == 0) EXIT
    Divisor = Divisor + 2                    ! not a divisor, try next
  END DO
  IF (Divisor*Divisor > Number) THEN ! divisors exhausted?
    Count = Count + 1                        ! yes, this Number is a prime
    WRITE(*,*) 'Prime number #', Count, ': ', Number
  END IF
END DO

WRITE(*,*)
WRITE(*,*) 'There are ', Count, ' primes in the range of 2 and ', Range
```

Factoring a Number: 1/3

- **Given a positive integer, one can always factorize it into prime factors. The following is an example:**

$$586390350 = 2 \times 3 \times 5^2 \times 7^2 \times 13 \times 17 \times 19^2$$

- **Here, 2, 3, 5, 7, 13, 17 and 19 are prime factors.**
- **It is not difficult to find all prime factors.**
  - **We can repeatedly divide the input by 2.**
  - **Do the same for odd numbers 3, 5, 7, 9, ….**
- **But, we said "*prime*" factors. No problem, multiples of 9 are eliminated by 3 in an earlier stage!**

Factoring a Number: 2/3

```
PROGRAM Factorize
  IMPLICIT NONE
  INTEGER :: Input
  INTEGER :: Divisor
  INTEGER :: Count

  WRITE(*,*) 'This program factorizes any integer >= 2 --> '
  READ(*,*) Input
  Count = 0
  DO                                      ! remove all factors of 2
    IF (MOD(Input,2) /= 0 .OR. Input == 1) EXIT
    Count = Count + 1                     ! increase count
    WRITE(*,*) 'Factor # ', Count, ': ', 2
    Input = Input / 2                     ! remove this factor
  END DO
  …… use odd numbers here ……
END PROGRAM Factorize
```

Factoring a Number: 3/3

```
Divisor = 3                         ! now we only worry about odd factors
DO                                  ! Try 3, 5, 7, 9, 11 ....
  IF (Divisor > Input) EXIT         ! factor is too large, exit and done
  DO                                ! try this factor repeatedly
    IF (MOD(Input,Divisor) /= 0 .OR. Input == 1) EXIT
    Count = Count + 1
    WRITE(*,*) 'Factor # ', Count, ': ', Divisor
    Input = Input / Divisor         ! remove this factor from Input
  END DO
  Divisor = Divisor + 2             ! move to next odd number
END DO
```

Note that even 9, 15, 49, … will be used, they would only be used once because `Divisor = 3` removes all multiples of 3 (*e.g.*, 9, 15, …), `Divisor = 5` removes all multiples of 5 (*e.g.*, 15, 25, …), and `Divisor = 7` removes all multiples of 7 (e.g., 21, 35, 49, …), etc.

Handling End-of-File: 1/3

- **Very frequently we don't know the number of data items in the input.**
- **Fortran uses `IOSTAT=` for I/O error handling:**

```
READ(*,*,IOSTAT=v) v1, v2, …, vn
```

- **In the above, `v` is an `INTEGER` variable.**
- **After the execution of `READ(*,*)`:**
  - **If `v = 0`, `READ(*,*)` was executed successfully**
  - **If `v > 0`, an error occurred in `READ(*,*)` and not all variables received values.**
  - **If `v < 0`, encountered end-of-file, and not all variables received values.**

Handling End-of-File: 2/3

- **Every file is ended with a special character. Unix and Windows use Ctrl-D and Ctrl-Z.**
- **When using keyboard to enter data to `READ(*,*)`, Ctrl-D means end-of-file in Unix.**
- **If `IOSTAT=` returns a positive value, we only know something was wrong in `READ(*,*)` such as type mismatch, no such file, device error, etc.**
- **We really don't know exactly what happened because the returned value is *system dependent*.**

Handling End-of-File: 3/3

```
INTEGER :: io, x, sum

sum = 0
DO
  READ(*,*,IOSTAT=io) x
  IF (io > 0) THEN
    WRITE(*,*) 'Check input. Something was wrong'
    EXIT
  ELSE IF (io < 0) THEN
    WRITE(*,*) 'The total is ', sum
    EXIT
  ELSE
    sum = sum + x
  END IF
END DO
```

input

```
1
3
4
```

output

```
The total is 8
```

input

```
1
&
4
```

no output

Computing Means, etc: 1/4

- **Let us compute the arithmetic, geometric and harmonic means of unknown number of values:**

arithmetic mean = $\dfrac{x_1 + x_2 + \ldots\ldots + x_n}{n}$

geometric mean = $\sqrt[n]{x_1 \times x_2 \times \ldots\ldots \times x_n}$

harmonic mean = $\dfrac{n}{\dfrac{1}{x_1} + \dfrac{1}{x_2} + \ldots\ldots + \dfrac{1}{x_n}}$

- **Note that only *positive* values will be considered.**
- **This naïve way is not a good method.**

Computing Means, etc: 2/4

```
PROGRAM ComputingMeans
   IMPLICIT NONE
   REAL    :: X
   REAL    :: Sum, Product, InverseSum
   REAL    :: Arithmetic, Geometric, Harmonic
   INTEGER :: Count, TotalValid
   INTEGER :: IO                   ! for IOSTAT=

   Sum        = 0.0
   Product    = 1.0
   InverseSum = 0.0
   TotalValid = 0
   Count      = 0
   …… other computation part ……
END PROGRAM ComputingMeans
```

Computing Means, etc: 3/4

```
DO
   READ(*,*,IOSTAT=IO) X  ! read in data
   IF (IO < 0) EXIT       ! IO < 0 means end-of-file reached
   Count = Count + 1      ! otherwise, got some value
   IF (IO > 0) THEN       ! IO > 0 means something wrong
      WRITE(*,*) 'ERROR: something wrong in your input'
      WRITE(*,*) 'Try again please'
   ELSE                   ! IO = 0 means everything is normal
      WRITE(*,*) 'Input item ', Count, ' --> ', X
      IF (X <= 0.0) THEN
         WRITE(*,*) 'Input <= 0. Ignored'
      ELSE
         TotalValid = TotalValid + 1
         Sum = Sum + X
         Product = Product * X
         InverseSum = InverseSum + 1.0/X
      END IF
   END IF
END DO
```

Computing Means, etc: 4/4

```
WRITE(*,*)
IF (TotalValid > 0) THEN
   Arithmetic = Sum / TotalValid
   Geometric  = Product**(1.0/TotalValid)
   Harmonic   = TotalValid / InverseSum
   WRITE(*,*) '# of items read --> ', Count
   WRITE(*,*) '# of valid items -> ', TotalValid
   WRITE(*,*) 'Arithmetic mean --> ', Arithmetic
   WRITE(*,*) 'Geometric mean  --> ', Geometric
   WRITE(*,*) 'Harmonic mean   --> ', Harmonic
ELSE
   WRITE(*,*) 'ERROR: none of the input is positive'
END IF
```

The End