DO WHILE Loop

FORTRAN-90 DO WHILE Statement

One of the important constructs is the loop construct.

The DO WHILE statement can be used to implement a loop construct in Fortran 90.

Syntax:

```
1.   •
2.   •
3.   •
4    DO WHILE (condition)
5.        •
6.        •
7.        •
8    END DO
9.   •
10.  •
```

Statements may be executed in the following order:

1 2 3 4 5 6 7 8 4 5 6 7 8 4 5 6 7 8 4 5 6 7 8 4 5 6 7 8 4 9 10 11 . . .

Loop Execution

The loop is PRE-tested: the condition check is made BEFORE going into loop. If the condition is initially false, no statements inside the DO WHILE are executed, execution continues with the first statement following the END DO.

If the condition is initially true, the statements inside the loop are executed (sequentially), Control is then transferred back up to the DO WHILE, the condition is checked again and the process is repeated. The loop stops when the condition is checked at the top of the loop and it is false, at which time control is transferred to the first statement following the END DO.

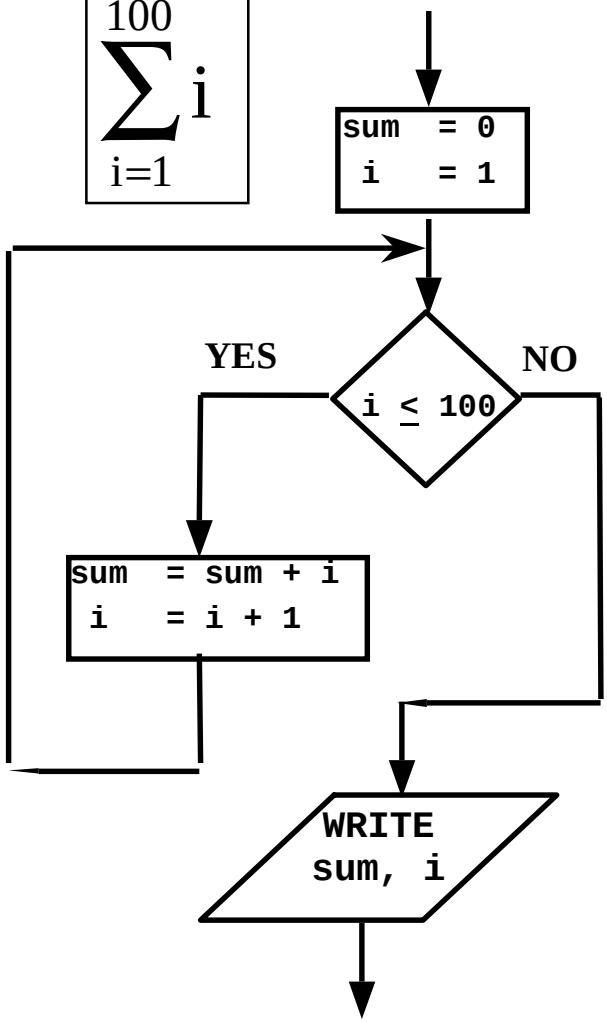
There must be at least one statement in the body of the loop that affects the value of the boolean expression, otherwise if the condition is true initially an infinite loop occurs.

DO WHILE Example

Problem: Determine the sum of the integers from 1 to 100.

$$\sum_{i=1}^{100} i$$

```
i = 1
sum = 0
DO WHILE ( i <= 100 )
 sum = sum + i
 i = i + 1
END DO
WRITE(*,*) sum, i
```



Trace

i	sum
1	0
2	1
3	3
4	6
5	10
6	15
•	•
•	•
•	•
101	5050

Note:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

Processing Methods

Three Common Data File Processing Techniques

1. End-of-File

   Performed with READ IOSTAT clause in FORTRAN

2. Trip Record or Sentinel Data Values

   Dummy Data Values

3. Data Header Value

   Initial Data Count

Trip Record or Sentinel Data Values

Requires the input data file to be terminated with a line, (the last line in the file), that contains dummy data values, (i.e. values that could NOT possibly occur, even by accident, in the input data.

Example Problem: Read a non-negative integer from each line until a negativeone (-1) is encountered. Count the number of integers between 0 & 9, 10 & 19, and 20 & 29.

```
      PROGRAM sample
      IMPLICIT NONE
      INTEGER :: n09 = 0, n1019 = 0, n2029 = 0, value
                                                  priming read
50    READ (9, 100) value
100   FORMAT (I4)
      DO WHILE (value .NE. -1)
         IF(( 0 <= value).AND.(value <= 9)) THEN
                n09 = n09 + 1
         ELSE IF ((10 <= value).AND.(value <= 19)) THEN
                n1019 = n1019 + 1
         ELSE IF ((20 <= value).AND.(value <= 29))THEN
                n2029 = n2029 + 1
         END IF
         READ ( 9, 100 ) value
      END DO
      STOP
200   WRITE (10,300) n09, n1019, n2029
300   FORMAT ('1',I3, ' between 0 and 9', i3, ....
      END PROGRAM sample
```

Processing Methods (cont)

Data Header Value

Requires the input data file to contain a single integer on the first line that represents a count of the number of lines/records, (usually all), that are to be read & processed, starting at line two.

Example Problem: Read in an input file that is set up for data header value processing, where each line contains one integer. Produce a sum of the lines specified by the data header value.

```
PROGRAM example
•    •    •
SUM = 0
READ (9,100) nlines
counter = 0
DO WHILE (counter < nlines)
     counter = counter + 1
     READ(9,101) data
     sum = sum + data
END DO
WRITE (*,*) sum
STOP
END PROGRAM example
```

Data File

```
 3
 5
-1
 6
 4
```

Trace

sum	counter	data
0	0	
5	1	5
4	2	-1
10	3	6

What would a read loop produce for sum here?

What would the sentinel method produce for sum here?

Named Constants

PARAMETER STATEMENT

Syntax:

Type, PARAMETER :: constname1= const1, constname2= const2...

- the parameter statement allows the programmer to define names for specific constant values, that relate to the problem, that does **NOT** change during program execution.

- is a specification statement (non-executable)

- **must** appear before the first executable statement

- the value of the constant name **cannot** be changed in the program, i.e., the constant name **cannot** appear on the left-hand side of an assignment statement or in a read statement.

Advantages of using PARAMETER statement with constant names:

- program is easier to modify for future considerations
- makes program easier to understand

Parameter Statement (example)

Payroll Calculation Program

```
PROGRAM parameter_example
  IMPLICIT NONE
  REAL,PARAMETER::payrate = 6.50, taxrate = 0.18
! payrate and taxrate cannot be altered
  INTEGER :: employee_id, ios
  REAL :: hours, gross, tax, net
  OPEN ....
  OPEN ....
  READ (9,101,IOSTAT = ios) employee_id, hours
101 FORMAT (I9, TR1, F5.2)
  DO WHILE (ios >= 0)
    gross = payrate * hours
    tax = taxrate * gross
    net = gross - tax
    WRITE (10,199)employee_id, gross, tax, net
199   format ( ' ', 'Employee ID:',I9, &
                  'Gross Pay:',F8.2,    &
                  'Tax Deducted:', F8.2, &
                  'Net Pay:',F8.2 )
    READ(9, 101, IOSTAT = ios) employee_id, hours
  END DO
```

Modifiability

In very large programs parameter constants allow code to easily updated when constant values do change. No text searching need be performed upon the code. Simply changing the PARAMETER values and recompiling accomplishes the update.

LOGICAL Variables Revisited

LOGICAL Variables

Variables that take on either of the 2 values:

.TRUE. or **.FALSE.**

Declare as:

LOGICAL :: vname1, vname2

Logical variables aids documentation of complex decisions

Increases program logic readability

Must be explained in the program documentation the same as all other variables

Logical variables can be read and written (using Lw format code) but we will not do so in this class.

Examples:

```
LOGICAL :: legal, valid, dummydata
INTEGER :: age, ssn
age = 41
ssn = -10
dummydata = .FALSE.
legal = ( age >= 21 )
valid = ( ssn > 0 )
dummydata = ( score == -1)
```

LOGICAL Variable Example

Problem:

Process at least 1 and at most 100 automobile parts and costs.

Input: part number (integer) and cost (must be positive real) with a sentinel of -1 in the part number position.

Output: count of the part numbers, total cost of all parts and number of bad data lines.

Note: part number will be valid BUT cost may be invalid and must be ignored when it is invalid. The program must process no more than 100 parts, watching for the sentinel.

Sample Input Data

```
12345 12345678
32145    34421
44231  -666666
  234     9995
 9188      -001
   -1      -100
 8871     8990
```

Ouput Results

```
Illegal cost, ignored-> -6666.66
Illegal cost, ignored-> -0.01
Summary: # parts total   # bad
                   cost    lines
           3   123900.95    2
```

LOGICAL Example (cont)

```
 PROGRAM parts
   IMPLICIT NONE
   INTEGER :: partnum, nparts = 0, badlines = 0
    REAL :: price, totcost = 0, cost
   INTEGER, PARAMETER :: sentinel = -1, maxparts = 100
   LOGICAL :: moredata
   OPEN (9, FILE = 'parts.dat' )
   OPEN(10, FILE = 'parts.rst' )
   moredata = .TRUE.
   DO WHILE ( moredata )
     READ(9, 100)partnum, cost
100  FORMAT(I5, TR1, F8.2)
     IF (partnum .NE. sentinel) THEN
        IF (cost > 0.0) THEN
           nparts  = nparts + 1
           totcost = totcost + cost
        ELSE
           WRITE(10,200) cost
200        FORMAT('Illegal cost, ignored->',F9.2)
           badlines = badlines + 1
         END IF
        moredata = (nparts < maxparts)
      ELSE
        moredata = .FALSE.
      END IF
  END DO
  WRITE (10, 201)
  WRITE (10, 202)
201 FORMAT ('Summary: # parts   total   # bad')
202 FORMAT ('                     cost   lines')
    WRITE (10, 203) nparts, totcost , badlines
203 FORMAT ( '     ', I3,TR5, F10.2, TR5, I3 )
  STOP
  END PROGRAM parts
```

Program Design

Problem: Determine if a positive integer is a perfect number.

A perfect number is a positive integer that is equal to the sum of its divisors, excluding the number itself; ex. 6, 28, 120, 496 . . .

Top-Down Design

1. Get the number to be tested
2. Determine the divisors of the number
3. Check if the divisors add up to the number.

    1.1 Prompt the user for the number (p)
    1.2 Read the number (check if number is positive int)

    2.1 set the divisor sum = 1
    2.2 check each integer, x, from 2 . . . p-1
    2.3 If the integer in 2.2 is a divisor add it to the sum

    3.1 If divisor sum = p then output p is perfect
        Else output p is NOT perfect

        2.2.1 Find the remainder of p/x
            2.2.1.1 remainder = p - truncate(p/x) * x
        2.2.2 If the remainder of p/x is 0 then x is a divisor

The “truncate” function used above just returns the result of doing an integer division with p and x — in other words, just doing a Fortran integer divide.

Program Implementation: Version 1

```
PROGRAM purenumber
IMPLICIT NONE
INTEGER::pernum, divnum, divsum, rem
LOGICAL::divsor, dataend , lastdiv , perfect
INTEGER, PARAMETER :: nomore = 0, initsum = 1, firstdiv = 2
WRITE(*,*)'Enter a possible perfect number or 0 to quit'
READ ( * , * ) pernum
dataend = ( pernum <= nomore )
DO WHILE ( .NOT. dataend )
     divnum = firstdiv
     divsum = initsum
     lastdiv = ( divnum == pernum )
     DO WHILE ( .NOT. lastdiv )
          rem = MOD ( pernum , divnum )
          divsor = ( rem == 0 )
          IF ( divsor ) divsum = divsum + divnum
          divnum = divnum + 1
          lastdiv = ( divnum == pernum )
     END DO
     perfect = ( divsum == pernum )
     IF ( perfect ) THEN
          WRITE(*,*)'The number ', pernum,' is PERFECT'
     ELSE
          WRITE(*,*)'The number ', pernum,' is NOT perfect'
     END IF
     WRITE(*,*)'Enter a possible perfect number or 0 to quit'
     READ ( * , * ) pernum
     dataend = ( pernum <= nomore )
END  DO
STOP
END PROGRAM purenumber
```

Efficiency Analysis

Processing Time

The bulk of the processing time in programs is spent in loops.

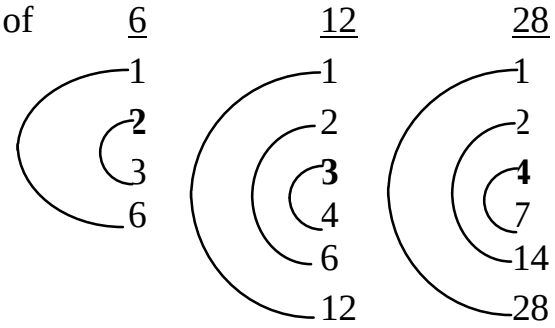
In the Perfect Number program, if a user enters a large number like 1,000,000 the inner while loop executes approx. 10^6 times.

Assume that on a very slow machine this takes 10 minutes to run.

In order to make the program faster we must be able to solve the problem while reducing the loop -> this requires knowing more about the problem.

Divisors:

List the Divisors of



Note:

- Divisors come in pairs
- The largest divisor excluding the number itself is $\leq$ 1/2 of the number
- ❁ Applying this to the perfect number program would cut the loop in half -> reducing execution time to 5 minutes

However, since divisors come in pairs, if one divisor is known both are known, thus the question becomes what is the relationship of 2 - 6, 3 - 12, 4 - 28?

Answer, they are all $\leq$ ¦number.

❁ Applying this to the perfect number program, the square root of 1,000,000 is 1,000 & since 1000 is 1/10000 of 10^6 this would reduce execution time to 600sec/1000 or 0.6 sec.!!!!

Program Update:Version 2

```
PROGRAM purenumber
IMPLICIT NONE
INTEGER :: pernum, divnum, divsum, rem
LOGICAL :: divsor, dataend , lastdiv , perfect
INTEGER, PARAMETER :: nomore = 0 , initsum = 1, firstdiv = 2
WRITE(*,*)'Enter a possible perfect number or 0 to quit'
READ ( * , * ) pernum
dataend = ( pernum <= nomore )
DO WHILE ( .NOT. dataend )
     divnum = firstdiv
     divsum = initsum
     lastdiv = ( divnum > INT(SQRT(REAL(pernum ))))
     DO WHILE ( .NOT. lastdiv )
         rem = MOD ( pernum , divnum )
         divsor = ( rem == 0 )
         IF (divsor) divsum= divsum + divnum + (pernum/divnum)
         divnum = divnum + 1
         lastdiv = ( divnum > INT(SQRT(REAL(pernum ))))
     END DO
     perfect = ( divsum == pernum )
     IF ( perfect ) THEN
         WRITE(*,*)'The number ', pernum,' is PERFECT'
     ELSE
         WRITE(*,*)'The number ', pernum,' is NOT perfect'
     END IF
     WRITE(*,*)'Enter a possible perfect number or 0 to quit'
     READ ( * , * ) pernum
     dataend = ( pernum <= nomore )
END DO
STOP
END PROGRAM purenumber
```