Computers in Engineering COMP 208

Indefinite Loops

Michael A. Hawker

Implicit Implied DO – LOOP

```
REAL :: A(1000)
INTEGER :: I, SIZE
READ(*,*) SIZE
READ (*,*) A
```

- Reads values sequentially like a regular do loop
- It must fill the entire array, not just the first SIZE values

Implied DO – LOOP

- In our ISBN example, we could input the digits as follows:

```
READ (*,*) (digits(I), I=1,10)
```

- We could input all of the digits on one or more lines separated by blanks
- The first 10 digits would be read and stored in the digits array

Implied DO – LOOP

```
REAL :: A(1000)
INTEGER :: I, SIZE
READ (*,*) SIZE
READ (*,*) (A(I), I=1,SIZE)
```

- Reads values sequentially from a line
- If there are not enough values on the line it starts a new line
- This is called an inline or implied DO loop

Why Use an Implied DO Loop

- Faster, Easier, and More Convenient
- Allows for easier Access to Change Number of Loops

```
REAL :: A(1000)
INTEGER :: I, SIZE
READ (*,*) SIZE
READ (*,*) (A(I), I=1,SIZE)
```

Compute Sum of Array Elements

```
REAL :: Data(100)
REAL :: Sum
. . .
Sum = 0.0
DO k = 1, 100
  Sum = Sum + Data(k)
END DO
```

Inner Product of Vectors

- The inner product of two vectors is the sum of the products of corresponding elements.

```
REAL :: V1(50), V2(50)
REAL :: InnerProduct
INTEGER :: dim, n
READ(*,*) dim          !actual dimension of vector
InnerProduct = 0.0
DO n = 1, dim
  InnerProduct = InnerProduct + V1(n)*V2(n)
END DO
```

Find Maximum Value

- How do we find the largest value in an array?
- Imagine a deck of cards that we look through one at a time
- Keep track of the largest value
- Start with the one on the first card
- Keep looking and note whenever a larger value is found

Find Maximum Value

```
PROGRAM FINDMAX
  IMPLICIT NONE
  INTEGER :: MARKS(210)
  INTEGER :: MAX, I
  READ(*,*) MARKS
  MAX = MARKS(1)
  DO I = 2, 210
    IF (MARKS(I) > MAX) MAX = MARKS(I)
  END DO
  WRITE (*,*) "THE HIGHEST MARK IS: ", MAX
END PROGRAM FINDMAX
```

Indefinite Iterators

- For some applications, we do not know in advance how many times to repeat the computation
- The loop will need to continue until some condition is met and then terminate

Indefinite Iterator

- The iterator we can use has the form

```
DO
   statement block, s
END DO
```

- The block, s, is evaluated repeatedly an indeterminate number of times

A Repetitive Joke

- Why did the Computer Scientist die in the Shower?

- The instructions on the shampoo label said:
  1. Rinse
  2. Lather
  3. Repeat

Infinite Loops

- A danger in using this construct is that the loop might never terminate.
- This loop computes the sum of a sequence of inputs

```
REAL :: x, Sum
Sum = 0.0
DO
  READ(*,*) x
  Sum = Sum + x
END DO
```

Terminating a Loop

- The general DO loop will go on forever without terminating
- How do we get out of it?
- The **`EXIT`** statement causes execution to leave the loop and continue with the statement following the **`END DO`**

Sum Positive Input Values

- Read real values and sum them. Stop when the input value becomes negative.

```
REAL :: x, Sum
Sum = 0.0
DO
  READ(*,*) x
  IF (x < 0) EXIT
  Sum = Sum + x
END DO
WRITE (*,*) "Sum is: ", Sum
```

GCD

- The greatest common divisor of two integers is the largest number that divides both of them
- There are numerous applications that require computing GCD's
- For example, reducing rational numbers to their simplest form in seminumeric computations
- We present a very simple (slow) algorithm

A GCD Algorithm

- The GCD is obviously less than or equal to either of the given numbers, x and y
- We just have to work backwards and test every number less than x or y until we find one that divides both
- We stop when we find a common divisor or when we get to 1

A Simple GCD Computation

```
PROGRAM gcd
   INTEGER :: x, y, g
   READ (*,*) x, y

   g = y
   DO
       IF (mod(x,g)==0 .AND. mod(y,g)==0) EXIT
       g = g - 1
   END DO

   WRITE (*,*) "GCD of ", x, " and ", y, " = ", g
END PROGRAM gcd
```

Finding Square Roots

- Newton presented an algorithm for approximating the square root of a number in 1669
- The method starts with an initial guess at the root and keeps refining the guess
- It stops refining when the guess is close to the root, that is when it's square is close to the given number

Finding the Square Root

```
! ------------------------------------------------------------
! Use Newton's method to find the square root of a positive number.
! ------------------------------------------------------------
PROGRAM SquareRoot
  IMPLICIT NONE
  REAL :: A, R, NewR, Tolerance

  READ(*,*) A, Tolerance

  R = A                                        ! Initial approximation
  DO
    NewR = 0.5*(R + A/R)                       ! compute a new approximation
    IF (ABS(R*R - A) < Tolerance) EXIT         ! If close to result, exit
    R = NewR                                   ! Use the new approximation
  END DO

  WRITE(*,*) " The estimated square root is ", NewR
  WRITE(*,*) " The square root from SQRT() is ", SQRT(A)
  WRITE(*,*) " Absolute error = ", ABS(SQRT(A) - NewR)
END PROGRAM SquareRoot
```

Exp(x)

- The exponential function can be expressed as an infinite sum:

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^i}{i!} + \ldots$$

- A program to approximate the value can compute a finite portion of this sum
- We can sum terms until the final term is very small, say less then 0.00001 (or any other tolerance we might choose)

Compute Exp(x) (preamble)

```
! ----------------------------------------------
! Compute exp(x) for an input x using the infinite series of exp(x).
! ---------------------------------------------------------

PROGRAM  Exponential
   IMPLICIT  NONE

   INTEGER          :: Count                  ! # of terms used
   REAL             :: Term
   REAL             :: Sum
   REAL             :: X
   REAL             :: Tolerance = 0.00001    ! Tolerance

   READ(*,*)  X
```

Compute Exp(x) (main part of program)

```
   Count = 1
   Sum   = 1.0
   Term  = x                            ! the second term is x
   DO
      IF (ABS(Term) < Tolerance)  EXIT
      Sum   = Sum + Term
      Count = Count + 1
      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))

END PROGRAM  Exponential
```