COMP 208
Computers in Engineering

Lecture 08

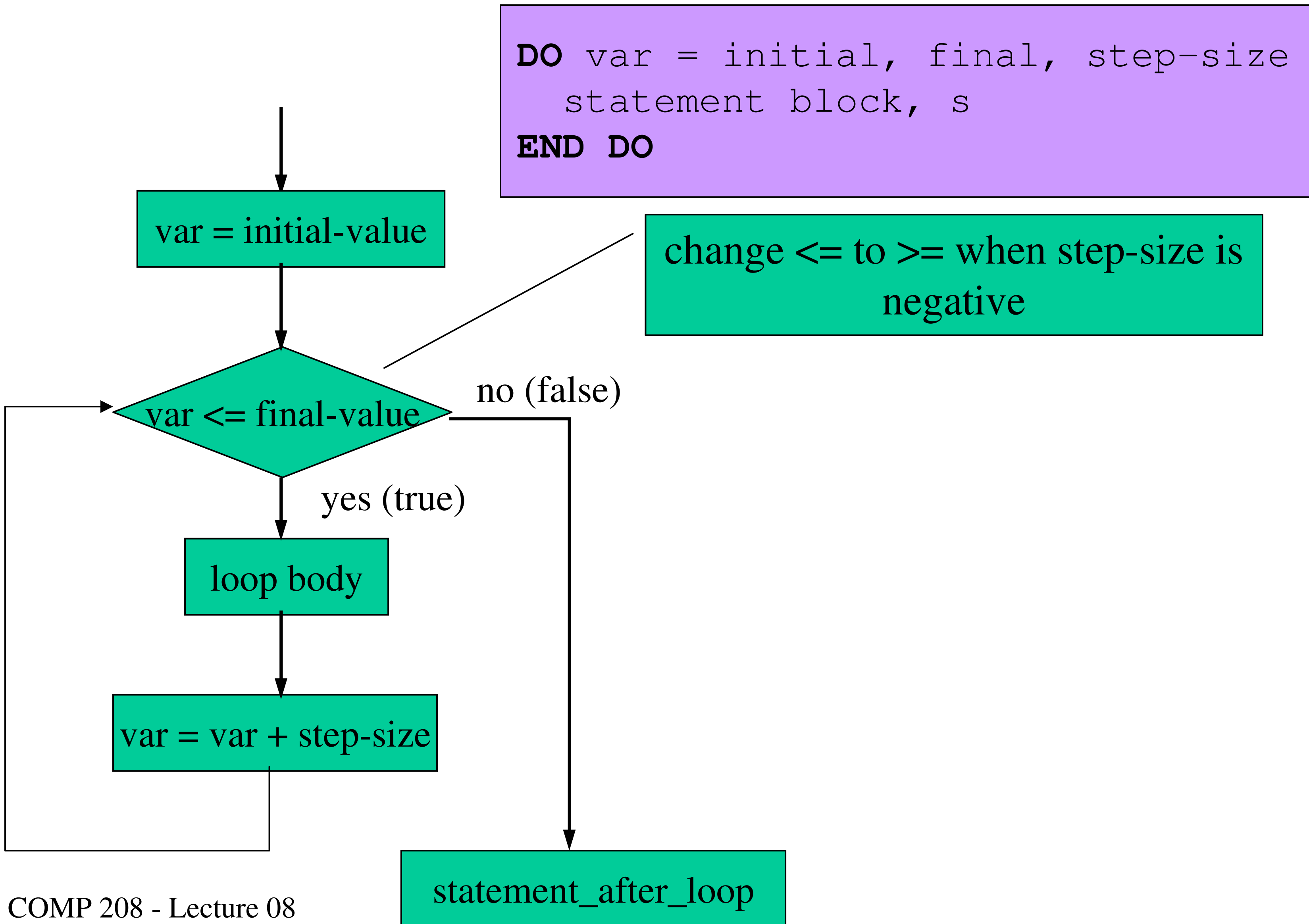
Jun Wang
School of Computer Science
McGill University

Fall 2007

Review

- Counter-controlled loop (definite iterator)

```
DO var = initial, final, step-size
  statement block, s
END DO
```



Example of counter-controlled loop

```
PROGRAM CountDown
  IMPLICIT NONE
  INTEGER :: counter

  DO counter = 3, 1, -1
    WRITE (*,*) "Counter: ", counter
  END DO
  WRITE (*,*), "Liftoff!"
  WRITE (*,*), "After loop, counter is: ", counter

END PROGRAM CountDown
```

```
Counter: 3
Counter: 2
Counter: 1
Liftoff!
After loop, counter is: 0
```

Example of counter-controlled loop

```
PROGRAM CountDown
  IMPLICIT NONE
  INTEGER :: counter

  DO counter = 1, 3, 1
    WRITE (*,*) "Counter: ", 4 - counter
  END DO
  WRITE (*,*), "Liftoff!"
  WRITE (*,*), "After loop, counter is: ", counter

END PROGRAM CountDown
```

```
Counter: 3
Counter: 2
Counter: 1
Liftoff!
After loop, counter is: 4
```

Example of counter-controlled loop

```
!============================================
! calculates 1 + 2 + 3 + 4 + 5
!============================================
PROGRAM SumOf5
  IMPLICIT NONE
  INTEGER :: sum, i

  sum = 0
  DO i = 1, 5
    sum = sum + i
  END DO

  WRITE (*,*), "The result is: ", sum

END PROGRAM SumOf5
```

Compound Data Structures

- We often want to process groups of data values in a uniform way
  - Digits in an ISBN
  - Grades in a class
  - Vector of real numbers
- Using individual variables is cumbersome
  - `INTEGER :: grade1, grade2, grade3, … , grade100`
- There is no way to uniformly examine or process the values stored in these variables

```
sum = sum + grade1
sum = sum + grade2
sum = sum + grade3
...
sum = sum + grade100
```

```
! We want something like the
! following. But it's not
! valid program
DO i=1, 100
  sum = sum + gradei
END DO
```

Arrays

- FORTRAN provides an array data type to support grouping related data together
- This allows them to be processed in a uniform way
- An array is a collection of data of the *same* type.
- The entire collection has a single name
- Individual values in the array are accessed by an INTEGER index

Declaring an Array

Syntax for an array declaration:

```
type :: name(bound)
```

Semantics

`type` is the type of the values that can be stored in each element of the array

bound specifies the range of indices for the subscript
Index of the array elements is between 1 and bound

In C, array index starts with 0

What happens when we declare an array?

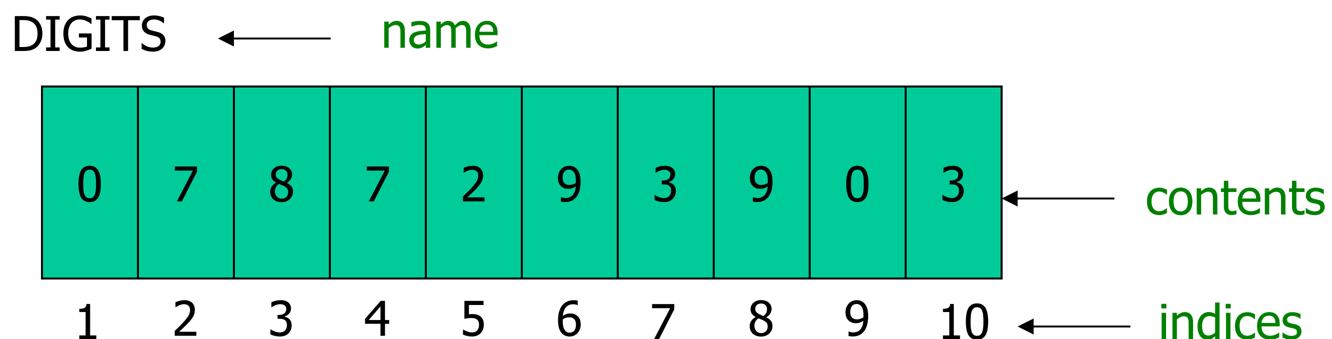
- When an array is declared, the computer allocates storage for a contiguous block of memory cells
- This block has the name we specify
- Each element in the block has the right size for holding values of the type that was specified
- The individual elements in the block can be referenced by an index
- The index starts at 1
- The index ranges up to the size we specify
- Each element can be treated as a simple variable of the specified type.

Declaring an Array

Example:

```
INTEGER :: DIGITS(10)
```

Visualizing an array:



How Do We Access the Elements?

To access individual elements in the array, we use the name of the array and the subscript (index) of the element

```
SUM = SUM + (11-POS)*DIGITS(POS)
```

Syntax:

```
array-name ( integer-expression )
```

Semantics:

array-name is the name of the array, and

integer-expression is an expression that evaluates to an integer. The value of this integer must be between 1 and the declared array size

Example

```
INTEGER :: grades(100)
...
DO i=1, 100
   sum = sum + grades(i)
END DO
```

```
INTEGER :: arr(3)
...
arr(1) = 3
arr(2) = 2
arr(3) = 1

WRITE (*,*) arr(1)
WRITE (*,*) arr(2)
WRITE (*,*) arr(3)
```

```
3
2
1
```

Out of Bounds?

- What happens if the index expression evaluates to 0? A negative number? A value greater than the array size?

- Who knows?

- What might happen:
  - Processor generates a run time error and stops (expensive to check)
  - Processor might just reference a memory cell near the array (dangerous)

Out of Bounds?

- What do I do?
  - Most compilers have a bounds checking option
  - Turn it on during testing
  - Turn it off when program fully developed to make execution more efficient

Using Arrays

- A natural mechanism for processing arrays is the DO-loop
- It allows us to go through and process each element in the array
  - e.g. increase all grades by 5%
- It also allows us to put values into the array to begin with

Initialize an Array to Zero

```
INTEGER :: UPPER = 100
INTEGER :: a(UPPER)
INTEGER :: i
DO i = 1, UPPER
  a(i) = 0
END DO
```

More on Arrays and Loops

Nathan Friedman

Fall, 2007

Input Values into an Array (DO – LOOP)

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

Since the READ statement has only 1 variable, A(I), it reads only one value. Therefore, the 1000 values must be entered in 1000 lines.

Input Values into an Array (DO – LOOP)

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

```
DO I = 1, 10
  READ (*,*) digits(i)
END DO
```

We would have to input 10 lines.

The first value on each line would be read

Input Values into an 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 waits for more values on a new line

This is called an inline or implied DO loop

Input Values into an Array (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

Input Values into an Array (Implicit Implied DO – LOOP)

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

Reads values sequentially like an implied do loop

It must fill the entire array, not just the first SIZE values

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 (logical expression-controlled loops)

- For some applications, we do not know in advance how many times to repeat the computation

- The loop continues until some condition is met and then it terminates

Indefinite Iterator

- Syntax

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

- The block, s, is evaluated repeatedly an indeterminate number of times
- The loop is terminated with an `EXIT` statement, usually when certain condition is true.

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

    !! Ensure both x, y are greater than 1
   IF (x < y) THEN
      g = x
    ELSE
      g = y
    END IF

   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

Newton's algorithm for square root

calculate the square root of 2

$x = 2$

estimate (R)	**difference ($R^2 - x$)**	**quotient (x / R)**	**average (R + x/R) / 2**
1	1	2/1=2	(1+2)/2=1.5
1.5	0.25	2/1.5 = 1.3333	(1.5+1.3333)/2=1.4167
1.4167	0.007	2/1.4167=1.4118	(1.4167 + 1.4118)/2 = 1.4142
1.4142	...	...	...

Newton’s Square Root Algorithm

```
! ------------------------------------------------------------
! Newton's method to find the square root of a positive number.
! ------------------------------------------------------------
PROGRAM SquareRoot
  IMPLICIT NONE
 REAL :: Tolerance = 0.001, A, R

  DO
    WRITE (*,*) "Enter a positive number:"
    READ (*,*) A
    IF (A >= 0.0) EXIT
  END DO

  R = A     ! Initial approximation
  DO
    IF (ABS(R*R - A) < Tolerance) EXIT  ! If close enough, exit
    R = 0.5 * (R + A/R)                 ! Update approximation
  END DO

  WRITE (*,*) "The estimated square root of ", A, " is ", R
  WRITE (*,*) "The square root from SQRT() is ", SQRT(A)
  WRITE (*,*) "Absolute error= ", ABS(SQRT(A) - R)

END PROGRAM SquareRoot
```

A Lame Joke

Why did the Computer Scientist die in the Shower?

The instructions on the shampoo label said:

1. Rinse
2. Lather
3. Repeat