Computers in Engineering COMP 208

Repetition and Storage

Michael A. Hawker

Repetition

- To fully take advantage of the speed of a computer, we must be able to instruct it to do a lot of work
- The program must be relatively short or it would take us too long to write
- To get the computer to do a lot of work, we must be able tell it to do some computations many times, perhaps with different data values each time

A Table of Values

- Problem: Output a table of numbers from 1 to 100 with their squares and cubes

```
1      1      1
2      4      8
3      9     27
4     16     64
    .   .   .
```

- We have to be able to repeat a computation over and over for the different numbers without writing 100 `WRITE` statements

A Table of Values

```
INTEGER :: Num

DO Num = 1, 100
  WRITE(*,*) Num, Num*Num, Num*Num*Num
END DO
```

Loops that Count

- The syntax of a definite iterator (often called a counted DO loop) is:

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

- var is an INTEGER variable called the control variable
- initial and final are INTEGER expressions
- step-size is an optional INTEGER expression. If omitted the default value is 1

Semantics of Counted DO Initialisation

- Evaluate the initial, final and step-size expressions.
  - These can be any expressions that give an integer value
  - They are evaluated only once before the loop is entered
- The step-size should not be 0.
  - The value if the step size is omitted is 1

Semantics of Counted DO (Counting Up)

- If the step-size is > 0, the loop counts up
  1. var = initial value
  2. If (var <= final value) then
     - Execute the statement block, s
     - var = var + step-size
     - Repeat step 2
  3. When var > final value, the loop ends and the statement after the END DO is executed

Semantics of Counted DO (Counting Down)

- If the step-size is < 0, the loop counts down
  1. var = initial value
  2. If (var >= final value) then
     - Execute the statement block, s
     - var = var + step-size (negative)
     - Repeat step 2
  3. When var < final value, the loop ends and the statement after the END DO is executed

- Output the odd numbers between 1 and 100, their squares and cubes.

```
INTEGER :: Num
DO Num = 1, 100, 2
   WRITE(*,*) Num, Num*Num, Num*Num*Num
END DO
```

Temperature Conversions

- Print a table of Celsius to Fahrenheit conversions:

```
INTEGER :: Celsius
REAL :: Fahrenheit

DO Celsius = -40, 40
 Fahrenheit = 1.8 * Celsius + 32.0
 WRITE(*,*) Celsius, " degrees Celsius = ", &
     Fahrenheit, " degrees Fahrenheit"
END DO
```

Note the negative initial value. Step size is 1.

```
INTEGER :: Celsius
REAL :: Fahrenheit

DO Celsius = 40, -40, -1
 Fahrenheit = 1.8 * Celsius + 32.0
 WRITE(*,*) Celsius, “ degrees Celsius = “, &
     Fahrenheit, “ degrees Fahrenheit”
END DO
```

Note the negative step size.

Average Value

- Input 1000 real numbers and compute the average value:

```
INTEGER :: Count, Number=1000
REAL :: Sum, Input
REAL :: Average

Sum = 0.0
DO Count = 1, Number
  READ(*,*) Input
  Sum = Sum + Input
END DO
Average = Sum / Number
```

Definite Iterator

- The DO loop we have looked at is called a definite iterator
- The body of the loop is executed a fixed number of times
- The control variable, `i`, takes on the values `x, x+s, x+2s, …, x+ks` where
  - `x` is the initial value,
  - `s` is the step size and
  - `x+ks` ≤ final value < `x+(k+1)s`

Processing Lists

- Counted do loops are used extensively in processing lists of data
- In the next application, we will see how to represent a list of data in a way that allows us to go through each value in the list using a do loop

ISBN Numbers

- ISBN numbers assign a unique identification number to every book published
- As with many such identification numbers, such as UPC codes, Postal Money Order serial numbers, Credit card numbers, there is a self checking code that allows us to reduce scanning and transmission errors

10 Digit ISBN Codes

An ISBN consists of 10 digits (newer standards will have 13 digits)

For example: **`0-7872-9390-3`**

1. The first digit is a country or language code
2. The next group of digits is the publisher
3. The next group is the item number
4. The final digit is a check digit

(The lengths of groups 2 and 3 may vary)

To calculate the check digit the International ISBN Agency specifies that

1. For each of the first nine digits, we multiply the digit by a weight depending on the position of the digit that goes from 10 down to 1
2. We then sum these products
3. The check digit is the number that, if added, will make this sum a multiple of 11

Verifying ISBN Numbers

```
PROGRAM isbn
   IMPLICIT NONE
   INTEGER :: digits(10)
   INTEGER :: pos, sum
   INTEGER :: check
   READ (*,*) digits
   sum = 0
   DO pos = 1,10
        sum = sum + (11-pos)*digits(pos)
   END DO
   check = mod(sum,11)
   IF (check == 0) THEN
        write(*,*) "ISBN is valid"
   ELSE
        write(*,*) "ISBN is invalid"
   END IF
END PROGRAM isbn
```

Verifying ISBN Numbers (with Logical Variables)

```
PROGRAM isbn
   IMPLICIT NONE
   INTEGER :: digits(10)
   INTEGER :: pos, sum
   LOGICAL :: valid
   READ (*,*) digits
   sum = 0
   DO pos = 1,10
        sum = sum + (11-pos)*digits(pos)
   END DO
   valid = mod(sum,11) == 0
   IF (valid) THEN
        write(*,*) "ISBN is valid"
   ELSE
        write(*,*) "ISBN is invalid"
   END IF
END PROGRAM isbn
```

- 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 :: digit1, digit2, digit3, … ,digit11`
- There is no way to uniformly examine or process the values stored in these variables

Arrays

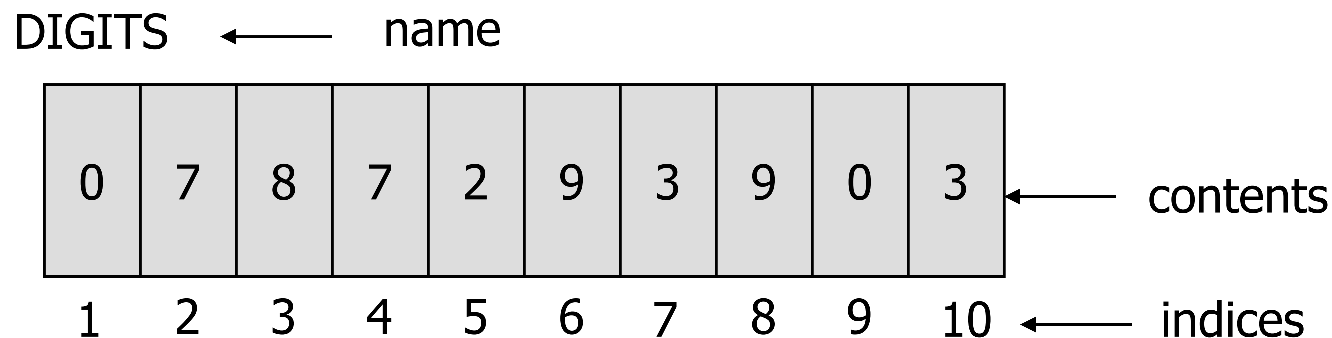
- FORTRAN provides an array data type to support grouping related data together
- This allows them to be processed in a uniform way
- An array is a collection of data of the same type.
- The entire collection has a single name
- Individual values in the array are accessed by an index

Declaring an Array

Example:

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

Visualizing an array:



Syntax for an array declaration:

```
type :: a(bound), b, c(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
- There is an array element of the specified type corresponding to each integer index between 1 and bound

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 cell in the block has the “shape” for holding values of the type that was specified
- The individual cells in the block can be referenced by an index
- The index starts at 1
- The index ranges up to the size we specify

How Do We Access The Cells?

To access individual variables in the collection, we use a subscript

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

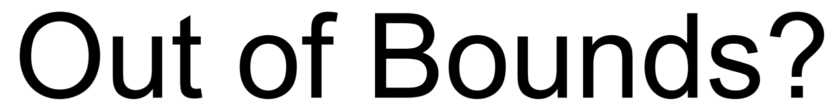
Syntax:

`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



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

- A natural mechanism for processing arrays is the DO-loop
- It allows us to go through and process each element in the array
- 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
```

Back to Counted Do Loops

- There are a few things we have to be careful about when using counted do loops

Don’ts of DOs

- Changing the values of the control variable or any variables involved in the controlling expressions of an iterator is risky.
- Some compilers will not allow this and will halt and signal an error. Others may allow it with unpredictable results.
- Programs should be portable. That is they should run on many different systems. Using features that are handled differently in different environments is not good.

Changing Changes…

Do not change the value of the *control-var*.

```
DO a = b, c
  a = b + c
END DO
```

Does the loop ever terminate?

```
DO a = b, c
  READ(*,*) a
END DO
```

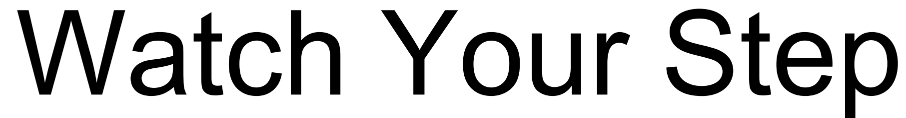
What does this do?

Warned…

- Do not change the value of any variable involved in *initial-value*, *final-value* and *step-size*.

```
DO a = b, d, e
  READ(*,*) b         ! initial-value changed
  d = 5               ! final-value changed
  e = -3              ! step-size changed
END DO
```

- The results are unpredictable!



Watch Your Step

- What happens if the step size is zero?

```
DO count = -3, 4, 0
...
END DO
```

- It seems to be an infinite loop.
- Some compilers might consider it an error and abort the program.

Input Values into an Array

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

- Reads one value per line
- (Each READ starts a new line)
- What happens if size is greater than 1000?

Input Values into an Array

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