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

Table of Odd Numbers

- 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.

Table in Descending Order

```
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)

The Check Digit

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
```


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

Arrays

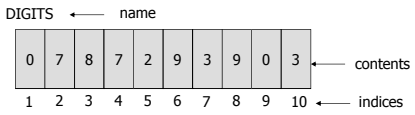
- FORTRAN provides an array data type to support grouping related data together
- This allows them to be processed in a uniform way
- An array is a collection of data of the same type.
- The entire collection has a single name
- Individual values in the array are accessed by an index

Declaring an Array

Example:

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

Visualizing an array:



Declaring 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

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
- 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

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
```


DO-WHILE

- DO ... WHILE loops are a special case used when a condition is to be tested at the top of a loop
- This is a looping structure provided in many different programming languages
- Syntax:

```
DO WHILE (logical expression)
  statement block, s
END DO
```


DO-WHILE

- Semantics:
  - Test the logical expression
  - If it evaluates to .TRUE., execute the statement block and go back to step 1.
  - If it evaluates to .FALSE., go to the statement after the END DO

DO-WHILE

DO-WHILE loops are equivalent to

```
DO
  IF .NOT.(logical expression) EXIT
  statement block s
END DO
```


Example

The DO loop of the program to compute exp(x) can be rewritten using a DO-WHILE

```
DO
  IF (ABS(Term) < Tolerance)  EXIT
  Sum   = Sum + Term
  Count = Count + 1
  Term  = Term * (X / Count)
END DO
```

```
DO WHILE (ABS(Term) >= Tolerance)
  Sum   = Sum + Term
  Count = Count + 1
  Term  = Term * (X / Count)
END DO
```


Warning!

- The loop only executes if the logical expression evaluates to .TRUE.
- If the value of this expression doesn't change, we will get an infinite loop
- The values of variables that the logical expression depends on must be modified within the loop
- (It still might not terminate, but at least we have a chance)

Nested DO-Loops

- A DO-loop can contain other DO-loops in its body.
- This nested DO-loop, must be completely inside the containing DO-loop.
- Note that an EXIT statement transfers control out of the inner-most DO-loop that contains the EXIT statement.

Nested DO-Loop Example

The outer loop has i going from 1 to 7 with step size 1.
For each of the seven values of i, the inner loop iterates 9 times with j going from 1 to 9.

```
INTEGER :: i, j
DO i = 1, 7
  DO j = 1, 9
    WRITE(*,*) i*j
  END DO
END DO
```

There are 63 values printed in total

Table of Exp(x) (preamble)

```
! ----------------------------------------------------------------
! This program computes exp(x) for a range of  values of x using the
! Infinite Series expansion of exp(x)
! The range has a beginning value, final value and step size.
! ----------------------------------------------------------------

PROGRAM  Exponential
   IMPLICIT  NONE
   INTEGER         :: Count
   REAL            :: Term
   REAL            :: Sum
   REAL            :: X
   REAL            :: ExpX
   REAL            :: Begin, End, Step
   REAL            :: Tolerance = 0.00001

   WRITE(*,*)  "Initial, Final and Step please --> "

   READ(*,*)   Begin, End, Step
```

Table of Exp(x) (body)

```
   X = Begin                     ! X starts with the beginning value
   DO
      IF (X > End)  EXIT         ! if X is > the final value, EXIT
      Count = 1
      Sum   = 1.0
      Term  = X
      ExpX  = EXP(X)             ! the exp(x) from Fortran's EXP()
      DO
         IF (ABS(Term) < Tolerance)  EXIT
         Sum   = Sum + Term
         Count = Count + 1
         Term  = Term * (X / Count)
      END DO

      WRITE(*,*)  X, Sum, ExpX, ABS(Sum-ExpX), ABS((Sum-ExpX)/ExpX)

      X = X + Step
   END DO
END PROGRAM  Exponential
```


GCD Revisited

- A more efficient way of computing the GCD of two integers is possible
- It doesn't even use division!!

Some GCD Facts

- The trivial cases:

  `gcd(k,k) = k , for nonzero k`

  `gcd(0,k) = gcd(k,0) = k, for nonzero k`
- The general case:

  `For i >= j, gcd(i,j) = gcd(i-j,j)`
- Using this, we can work backwards from the general case by reducing the larger of the two arguments until we reach one of the trivial cases

A GCD Program

```
INTEGER :: I, J, G
DO WHILE (I /= 0 .and. J /= 0 .and. I /= J)
  IF (I>J) THEN
    I = I - J
  ELSE
    J = J - I
  END IF
END DO
IF (I == 0) THEN
  G = J
ELSE
  G = I
END IF
```


Verifying ISBN Numbers

```
program isbn
   implicit none
   integer :: digits(10)
   integer :: pos, sum
   logical :: valid
   READ (*,"(10I1)") (digits(pos), pos = 1,10)
   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
```